A tabular interface for automated verification of event-based dialogs

Hung-Ming Wang
Gregory Abowd[a]

28 July 1994
CMU-CS-94-189

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

[a]College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

This work is the result of an independent study project within Carnegie Mellon's Master of Software Engineering (MSE) program. G. Abowd was a member of the MSE faculty and was sponsored in part by the U.S. Department of Defense when working on this project. G. Abowd is currently an Assistant Professor at Georgia Institute of Technology. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the DoD or the U.S. government.

Keywords: Computation Tree Logic, Dialog Analysis, Model Checking, Propositional Production System, Symbolic Model Verifier

Abstract

In this report, we investigate the feasibility of a tabular interface for the specification and analysis of event-based dialogues. These dialogues are used to define high-level descriptions of interactive systems, and they are based on Olsen's Propositional Production System (PPS) notation. The simulation of the abstract user-system dialogue is an effective means of matching a design with an expected task model. Monk and Curry have produced a prototype dialogue simulation tool, the Action Simulator, which demonstrates how a tabular paradigm can be used to specify and simulate the dialogue. Further analysis of the dialogue can uncover problems which are not so easy to detect with simulation. If we view the dialogue as defining a finite state machine, then we can make use of powerful model verification tools, such as Clarke's Symbolic Model Verifier (SMV) tool, to perform more powerful analyses on the dialogue. We also find that the tabular paradigm for input is an interesting alternative to the input language for the SMV tool. We provide in this report a mechanical translation from the tabular dialogue specification into SMV and provide templates or heuristics for various reachability analyses using the Computation Tree Logic (CTL) formalism. This research, therefore, presents the beginnings of two significant contributions. For the HCI community, we show how model verification tools can be used to provide a more powerful analytic technique beyond simulation for the specification and design of interactive dialogues. For the model verification community, we demonstrate the possibility of developing a simpler interface to specify and analyze certain finite state machines.

1. Introduction

The design of any complex system is an incremental process. The designers have coarse ideas at first. They must try out these ideas and refine them until a complete design specification is obtained. It is not cost effective to implement every design idea, so cheaper, more abstract models are built to explore early design ideas. A model may take the form of English prose descriptions, sketches of screens, executable prototypes, formal specifications, or user manuals. Whatever form it takes, a good model is:

- easy to describe, or specify;
- easy to understand and communicate its meaning;
- readily analyzable to determine its properties; and
- easily modified.

There are usually conflicts among the four desired attributes identified above. Informal textual descriptions might reduce the effort for description and communication, but they are difficult to reason about. More formal models have the advantage of applying a large body of mathematical knowledge to proof and analysis, but they are typically too cumbersome and difficult for use by average programmers. Prototyping can provide animated simulations of expected system behavior, helping to uncover problems before implementation, but it relies on human interaction with the prototypes and cannot automatically verify certain properties of the model.

In this report, we will examine the event-based dialogs which are used to model interactive behavior between a user and system. These dialogs can be defined at a very high level of abstraction for use in early interactive system design. Specifically, we will be concerned with Olsen's Propositional Production System (PPS) [2]. Monk and Curry have defined a tabular method for expressing a proper subset of PPS and have shown how it can be implemented on a standard spreadsheet package to provide a simulation of the dialog [3]. The tabular interface to PPS has several advantages. It is simple to define and understand the dialog, and possible to validate the model for correspondence to high-level task requirements in the form of usage scenarios. The dialog can easily be changed to suit the task scenarios. It is not possible, however, to do more sophisticated kinds of dialog analysis, such as determining if certain dialog states are reachable, or if certain actions are always reversible. Olsen, Monk and Curry have started to address these deeper analysis techniques, but have not yet shown how a more powerful analysis technique can be added to the expressive and simple tabular specification [6].

The main contribution of our work is to demonstrate how existing model verification tools can be linked to the tabular PPS dialog description to provide more powerful analyses. Model verification is an area of research of increasing importance in software engineering. Its basic premise is that a system, once described as a finite state machine, can be subjected to exhaustive analysis of its entire state space to determine what properties hold. The main drawback of this approach is the state explosion problem, in which the number of states of the system increases exponentially as it becomes more and more complex. Since realistic examples of system end up having very large state spaces, exhaustive search was considered too expensive. The main advances in model verification technology now allow for examination of systems with huge state spaces (over 10^{30} states) [8]. We will demonstrate how a tabular PPS event-based dialog can be translated into one particular model verification tool, the Symbolic Model Verifier (SMV), developed at CMU [4].

The link between tabular event-based dialog specification and model verification which we establish in this report represents two distinct contributions. First, dialog modeling is important in the design and analysis of interactive systems, but there are very few tools or techniques for doing this. Pre-condition and post-condition semantics of low-level interaction widgets have been used in the UIDE system to automatically generate an interface from its specification, but this does not speak to the analysis of the specification [2]. Probably the closest related technique to one we are proposing would be the Statemate system [13] based on Harel's statecharts [11][12]. Statecharts provide a visual formalism for specifying complicated state transition systems and the Statemate system provides a way to simulate the specification and perform certain reachability and deadlock checks on the specification. At this point, we are unable to say how our approach with model verification and tabular specification compares to statecharts and Statemate.

The second contribution of this work is the development of better interfaces for model verification technology. The advances in model verification that have enabled the efficient examination of complex systems with many states have not been accompanied by suitable advances in the interfaces to those tools. One of the major obstacles preventing their more widespread adoption is that the tools are too difficult to use. By demonstrating the mechanical translation from the tabular event-based dialog description into the SMV tool, we are laying the foundation for an enhanced interface to SMV which should prove more usable for certain applications.

The remainder of this report is organized as follows. Section 2 describes the overall framework of our method and clearly indicates how our work translating PPS tabular descriptions into SMV for further analysis fits into an overall design life cycle. In Section 3, we provide an overview of PPS and define the tabular definition of PPS dialog models. In Section 4, we provide a brief overview of the Computation Tree Logic (CTL) and the SMV tool. Section 5, the first significant section of this report, explains the translation step from a PPS tabular description into the CTL representation of SMV. In that section, we define the mechanical steps for translating from PPS into SMV and prove that the translation is well-defined. Section 6 presents a case study of analysis and gives a guideline of how certain desired properties can be cast in CTL formulae. Section 7 describes the follow-up activities if a property does not hold in the automatic analysis. Section 8 concludes our results and discusses the future work.

2. The Overall Process of Dialog Design

This section describes the overall dialog design process and the role that both PPS and SMV play in that process. The method we are advocating is divided into five steps, as shown in Figure 1 – formulating the dialog model in PPS, validating the PPS task model by simulation, mechanically translating the dialog model into SMV, automatically verifying CTL-based properties, and visualizing the design problems for correction.

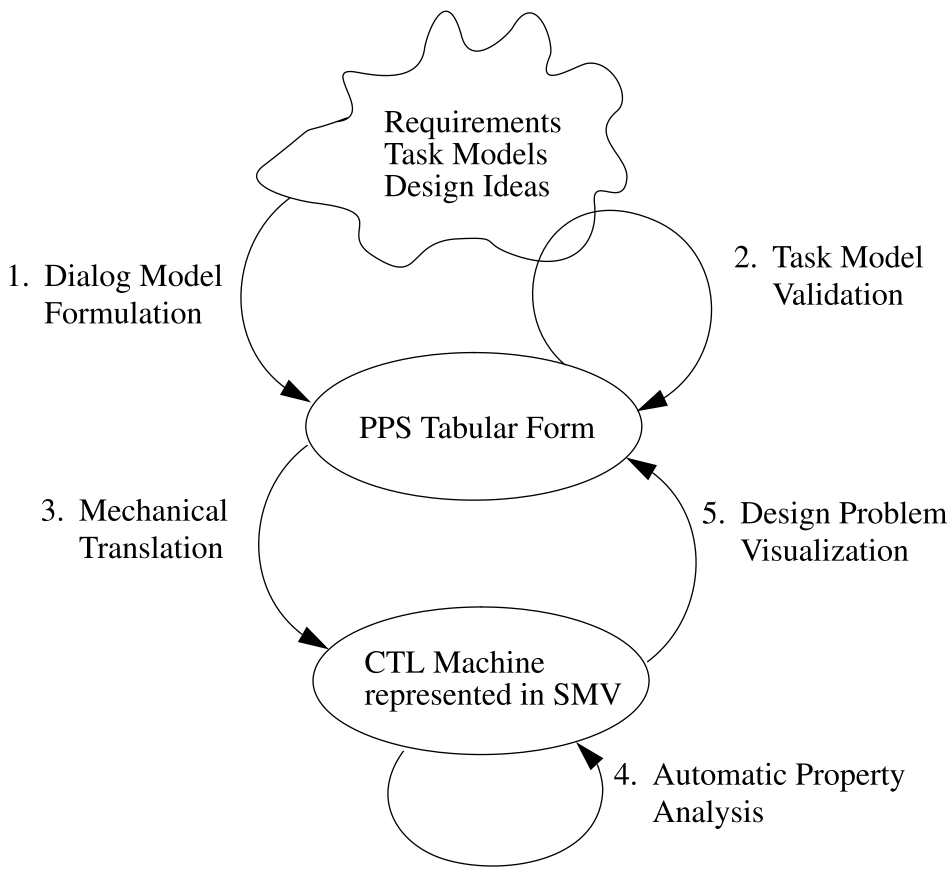

Figure 1. The dialog design process

2.1. Dialog Model Formulation

The first step in dialog design is to create an initial dialog description based on an understanding of the system to be built and the tasks it must support. Developing an adequate task model for use in initial design of an interactive system is a rich area of research called task analysis (for a summary of task analysis approaches, see Chapter 7 of [1]). Monk & Curry have developed a scenario-based method for generating a dialog model. There are several artifacts which result from their method (a work objective decomposition, descriptions of illustrative scenarios, and exceptions lists), but the principal artifact is a tabular dialog description in a reduced PPS syntax using their own tool, Action Simulator, a specialized spreadsheet package [3]. The work in this paper does not address this issue of dialog formulation any further. Rather, we are more concerned with how a dialog description can be manipulated and analyzed.

2.2. Animated Simulation against Task Model

The development of a good dialog model requires an iterative process. Once we have a PPS tabular specification, it is not difficult to evaluate the dialog model against the task model if there is some computer support. Simulation of the dialog is one way to check its adequacy. In Monk & Curry's work, the Action Simulator is used to compare a dialog model against the task scenarios used to develop it. This requires the use of a software program that keeps a record of what conditions are set and uses this record to display a list of user events that are available for selection. The designer can examine the effect of one of these events by selecting it. This changes the dialog state and a new list of available user events is displayed. The designer can run though the scenarios in the task model and check that the model allows the user to generate the appropriate user events and whether a task can be accomplished.

Analysis of a dialog model by observing its behavior in this way is useful to gain an understanding of the design. However, there are several questions a designer might ask that are tedious or difficult to answer by exhaustive simulation of the dialog itself. For example, Olsen, Monk & Curry [6] list several varieties of reachability properties that would be virtually impossible to check using the Action Simulator.

2.3. Mechanical Translation

Olsen, Monk & Curry also propose some algorithms to perform the reachability checks on a general PPS dialog, but they do not provide a tool for performing the analysis. Our aim in this report is to examine to what extent those more sophisticated analyses can be achieved by model verification technology that already exists. The model checking technique is very promising because the analysis can be automated and the kinds of properties that can be verified are very rich. If the PPS tabular specification were mechanically translatable into a form acceptable for model checking, then we would be able to take advantage of those tools.

A large part of this report is devoted to the transformation process from a PPS specification into the SMV input language. Both PPS and SMV are finite state systems. However, there are differences between the two formalisms. SMV supports the Computation Tree Logic (CTL), a branching-time temporal logic and it enforces that each branch of the computation tree must be infinite. PPS dialogs can have deadlocks. PPS have a collection of user events participating in the dialog state transitions but SMV must encode the user events as state information. These issues should be carefully addressed in the translation process.

SMV is a tool which can accept a CTL machine and CTL specifications. Based on our results, it should be straightforward to write a translator converting a standard PPS table into SMV. The translation process could then be hidden from the dialog designer.

2.4. Automatic Property Analysis

There are certain frequently desired properties a designer might want to assure before implementation. It is considered a poor design if there are points in the dialog where the system might deadlock, if tasks are unreasonably difficult to complete, or if effects cannot be undone. Being able to detect such potential usability problems early in the design process is a considerable advantage.

Having a CTL machine represented in SMV language translated from a PPS dialog, the designer can start to ask questions as CTL formulae which are automatically verified. However, it would be useful if we could identify the kinds of questions that are usually asked, and provide guidelines for how to express the questions in CTL formulae. We summarize several categories of questions and show how these questions can be cast in CTL logic, based on the properties defined by Olsen, Monk & Curry [6].

2.5. Design Problem Visualization

The SMV model checker will try to provide a counterexample if the property being checked is false. These counterexamples are output in a log file and relatively difficult to trace, especially when the state space is very large and complicated. We can redirect the log file output by SMV to drive the animation tool. In this way, the designer can easily visualize why and how the desired property fails, instead of tediously tracing the log file and often getting lost.

3. Propositional Production System

3.1. PPS Dialog Model

Propositional Production Systems were originally introduced by Olsen and used to specify human-computer dialog at a much higher level of abstraction for analysis purpose of developing verification algorithms [6]. However, production systems have a long history of use for modelling human-computer dialog. Tools for the generation of user interfaces use rules with pre- and post-conditions to describe the behavior of an interface, and automatically generate a user interface from the rule specification [2]. These tools have as their objective the low level specification of human-computer dialog. For our purpose, it does not matter whether the propositional production systems are used at a higher abstraction level (such as *"select delivery record N"*) or in a low level physical specification (such as *"release the left button"*). We simply treat PPS as an easy-to-write input language for model checking.[1]

In a PPS specification, the designers identify a set of user events and several fields. A field is

1. It is generally recognized that modelling at a more abstract level can enforce attention on critical issues in the early development phase. It is easier to verify a dialog model against user scenarios if the dialog model is at the same level of abstraction as the task model.

associated with a set of distinct values. At any time a field can only be in one of the values associated with it. That is, the values associated with a field are mutually exclusive so setting a new value unsets the value currently set. The vector of current values of all fields represents the current state of the dialog model. The designers also define a starting state vector and a collection of rules. A rule takes the following form:

user event, pre-condition vector, post-condition vector

A rule attaches a pre-condition vector and a post-condition vector to a user event. The pre- and post-condition vectors are written in propositional logic formulae in terms of fields. A pre-condition vector specifies when a rule is enabled; a rule is enabled if the current state of the dialog model makes the pre-condition vector of that rule true. A user event is enabled if the rule is enabled. Depending on the current state, there are probably zero, one, or more enabled user events. Each time a user can select one user event to execute from the set of enabled user events. The rule of the selected user event then fires. The post-condition vector of the firing rule changes the state of the dialog model, which becomes the conditions for the next rule to fire.

3.2. PPS Copier Example

Table 1 shows a PPS specification example of a copier taken from [3]. In this case, each field is associated with binary values but in general it is not necessary. The collection of rules are described using a tabular form. Each rule takes one row and has a user event name associated with it. The top line of each row specifies the pre-condition vector and the bottom line of each row specifies the post-condition vector. If a field is blank in a pre-condition vector, this indicates that we don't care about the value of that field when deciding if the rule is enabled. If a field is blank in a post-condition vector, this indicates that the value of that field will not be changed if the rule fires. With the starting state of *(Ready=NotOK, Copying=Off, OneCopy=Yes, DefSettings=Yes)*, only rule 1 is enabled and thus the user can only choose the user event *SwitchOn* to execute. After firing rule 1, the state becomes *(Ready=OK, Copying=Off, OneCopy=Yes, DefSettings=Yes)*, and therefore rules 2, 3, 5, and 7 are now enabled. This means that the user events *SwitchOff*, *Copy*, *MultipleCopies*, and *ChangeSettings* are enabled and the user can select any one of the four events to carry on. The selection is made by the user and hence the state of the dialog model is essentially determined by both the rules coded in the table and the user's selection.

Fields Ready: { OK, NotOK }
Copying: { On, Off }
OneCopy: { Yes, No }
DefSettings: { Yes, No }
Initial state : (Ready = NotOK, Copying = Off, OneCopy = Yes, DefSettings = Yes)

Rule	user event	Ready	Copying	OneCopy	DefSettings
1	SwitchOn	NotOK OK			

Rule	user event	Ready	Copying	OneCopy	DefSettings
2	SwitchOff	OK NotOK	Off		
3	Copy	OK	Off On		
4	FinishedCopying	OK	On Off		
5	MultipleCopies	OK	Off	Yes No	
6	CancelCopies	OK	Off	No Yes	
7	ChangeSettings	OK			Yes No
8	CancelSettings	OK			No Yes

Table 1. PPS specification of a copier

3.3. Expressiveness of PPS Description

A PPS specification of this kind is relatively easy to read and write. PPS is similar to a state transition diagram in that it consists of rules specifying possible state transitions. State transition diagrams are a representation most programmers are familiar with. The problem with using a simple state machine to model a complex system is that the large number of parallel options leads to a state explosion that makes specification and analysis intractable. PPS resolves this problem by working at the granularity of state sets instead of every single state.

Notice that the pre- and post-condition vectors and the starting state vector do not have to be a full vector. The absence of a value of a field in a vector simply means "don't-care" in a starting state (actually the set of starting states is more correct), means "don't-care" in a precondition, or means "don't-change" in a post-condition. In this way, PPS avoids the necessity of enumerating the entire state space. A key to the expressiveness of a PPS specification is the use of incomplete condition vectors to describe a wide variety of possible states. For a large system, the number of rules needed in a PPS specification is much less than the number of transitions needed in a typical state machine specification.

In addition, our PPS model also allows non-determinism to enhance its expressiveness. At any time during the course of the dialog, there is a set of enabled rules. It is possible to let multiple rules have the same event label, and therefore it is possible to have multiple rules with the same event label being enabled simultaneously. Given the same event label, any enabled rule with that event label can fire but only one can fire. The decision of which one to take is non-deterministic. It is arguable that in the high level design, it is a desired feature to allow non-determinism. The designer can specify the essential execution paths in a deterministic way while ignore others. Besides, the CTL model we adopt for automated verification well supports non-determinism.[1]

3.4. Formalism of PPS

Suppose N is the number of rules in a PPS specification, and each rule i is in the form of $(a_i, pre_i, post_i)$. Formally, a PPS system is a triple $PPS = (A, \Sigma, T)$ where

- A is a finite set of event labels; that is, $A = \{ a_i \mid 1 \le i \le N \}$.
- Σ is a finite set of dialog states.
- T is a binary relation where $T \subseteq A \times (\Sigma \to \Sigma)$, in which each member specifies one rule; that is, $T = \{ (a_i, trans_i) \mid 1 \le i \le N \}$.

We denote the domain of $trans_i$ by pre_i and the range of $trans_i$ by $post_i$. Each pair of $(pre_i, post_i)$ in a rule is actually a partial function $trans_i$ of signature $\Sigma \to \Sigma$, which maps a full pre-state vector to a full post-state vector.

4. Model Checking Technologies

4.1. Computation Tree Logic

Temporal logic and model checking have been used to verify properties in hardware systems. To apply model checking techniques, one needs to transform the system to be verified into an appropriate structure that the model checking tool can accept. One then specifies the property needed to be ensured in a logic formula and the tool automatically analyzes the structure and tells whether the property holds in the system. The specification language is a propositional, branching-time temporal logic called CTL (Computation Tree Logic).

The semantics of CTL formulae can be understood with respect to a labeled state-transition graph. Formally, suppose that AP is the underlying set of atomic propositions, we can define a CTL machine to be a triple $CTL = (S, R, P)$ where

- S is a finite set of states.
- R is a binary relation on S ($R \subseteq S \times S$) which gives the possible transitions between states and must be total; that is, $\forall\, x \in S \bullet (\exists\, y \in S \bullet (x, y) \in R)$.
- $P: S \to 2^{AP}$ assigns to each state the set of atomic propositions true in that state.

A *path* is an infinite sequence of states $(s_0, s_1, s_2, ...)$ such that $\forall\, i \bullet (s_i, s_{i+1}) \in R$. For any machine $CTL = (S, R, P)$ and any state $s_0 \in S$, there is an *infinite computation tree* with root labeled s_0 such that $s \to t$ is an arc in the tree iff $(s, t) \in R$. Figure 2 shows a CTL machine and the associated computation tree rooted at s_0.

1. The situation of having two enabled rules simply means two branching paths. We can construct a CTL machine where both paths exist in the computation tree.

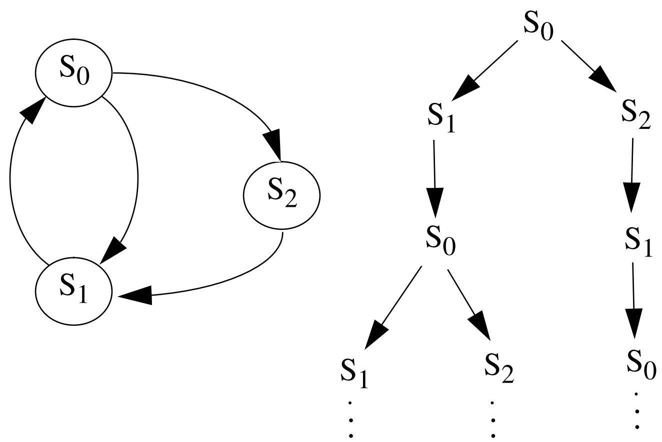

Figure 2. CTL machine and the corresponding tree for starting state *S_0*

CTL operators permit explicit quantification over all possible futures. The syntax and semantics for CTL formulae are defined in [7] and are only summarized as follows:

- Every atomic proposition is a CTL formula.
- If f and g are CTL formulae, then so are $\sim f$, f & g, $f \mid g$, $f \rightarrow g$, $AX\,f$, $EX\,f$, $A[f\;U\;g]$, $E[f\;U\;g]$, $AF\,f$, $EF\,f$, $AG\,f$, $EG\,f$.

The symbols ~ (not), & (and), and | (or) are logical connectives. These connectives and their derivable propositions, such as $f \rightarrow g$ (f implies g), have their usual meanings. X is the *next-time* operator. The formula $AX\,f$ ($EX\,f$) intuitively means that formula f holds in every (in some) immediate successor of the current state. U is the *until* operator, and the formula $A[f\;U\;g]$ ($E[f\;U\;g]$) intuitively means that along every (some) computation path there exists an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix. The formula $AF\,f$ ($EF\,f$) means that along every (some) path there exists some future state in which f holds. The formula $AG\,f$ ($EG\,f$) means that f holds in every state along every (some) path.

Taking the above copier example, perhaps we will want to check if the copier can be switched off at any time when the copier is on. The corresponding CTL formula would look like follows.

```
AG(Ready = OK -> EX(Ready = NotOK))
```

4.2. Symbolic Model Verifier

SMV is a model checker which accepts a CTL machine and a CTL formula, and automatically tests whether or not the formula holds in the machine. If the SMV model checker determines the formula is true, then the property holds in the CTL machine and also in the system from which the CTL machine is translated.

The primary purpose of the SMV input language is to describe the transition relation of a CTL machine. A detailed description of the syntax and semantics of the SMV input language, and the function of the SMV model checker can be found in [4].

If any of the specifications does not hold in the CTL machine, the SMV model checker will attempt to produce a counterexample, proving that the specification is false. Some formulae require infinite execution paths as counterexamples. In this case, the model checker outputs a looping path up to and including the first repetition of a state. However, generating a counterexample is not always possible, since formulae preceded by existential path quantifiers cannot be proved false by showing a single execution path.

SMV model checker needs to exhaustively search the state space. The BDD-based (Binary Decision Diagram) symbolic model checking algorithm makes it possible to efficiently determine whether specifications expressed in CTL formulae are satisfied [?]. We will not explore this topic as it is far out of the scope of this report.

5. Translation of Tabular Description into SMV

5.1. Mimicking User Events

In a PPS specification, each rule is associated with a user event. A user event is enabled if the current state makes the pre-condition vector of the rule true. At any time, the number of the enabled user events could be zero or more. If there are zero enabled user events, this means the dialog enters into a deadlock, and the user cannot do anything. If there are more than zero enabled user events, then the user can select one of them to execute. Each time after a rule fires, the current state is changed according to the post-condition vector of the rule, and a new collection of enabled user events should be determined for user's next selection.

The above description will probably give readers a dynamic impression as someone executes the PPS dialog. Essentially, after a PPS table is defined, a state machine is already statically defined. Given any state, the collection of enabled user events from that state can be easily computed by examining each rule. Consequently, if we treat a user event name as an input symbol, a user event name is simply like an alphabet accepted by FA (finite automaton).

More precisely, a user event name is like an alphabet accepted by NFA (non-deterministic finite automaton) because NFA allows (i) zero, (ii) one, or (iii) more transitions from a state on the *same* input symbol. Case (i) means that at a dialog state, a particular user event *E* is not enabled, and thus the user cannot select it. Case (ii) means that at a dialog state, a particular user event *E* is enabled, and there is only one rule making the user event *E* enabled. Case (iii) means that at a dialog state, a particular user event *E* is enabled, and there is more than one rule making the user event *E* enabled. Case (iii) is allowed because we allow multiple rules to have the same event label. This problem is dealt with in Section 5.2.

The problem of using a CTL machine to mimic a PPS system is that the CTL model does not include the concept of alphabets explicitly. The transition relation is specified by a formula which confines the relationships of the state vectors between two states; there exists a transition between two states if the two state vectors together make the formula true.

To model the user events, our strategy is to group all possible user event names of a PPS table into an event type, and declare a special state variable, `Event`, whose value always denotes an enabled user event. This implies that the "event value" has become part of the state information. This also implies that in the resultant CTL machine, the corresponding computation tree will only contain nodes each of whose `Event` variable represents an enabled user event; any other node (i.e., a dialog state with an non-enabled user event, or simply an unreachable dialog state) should not emerge in the computation tree.

Also, it is possible to have a deadlock in a PPS specification. In a CTL machine, however, each branch of the computation tree has an infinite trace. The deadlock modelling problem is dealt with in Section 5.3.

5.2. Non-determinism

There are two ways to specify transition relations of a CTL machine and its initial states in SMV. They can be specified by a collection of parallel assignments, introduced by the `ASSIGN` statements, or by propositional formulae directly, introduced by the `TRANS` and `INIT` statements.

An `ASSIGN` statement usually involves lots of case expressions. The value of a case expression is determined by the *first* expression on the right hand side of a colon such that the condition on the left hand side is true. This means that if we use this way to specify transition relations, only one rule will fire when the same user event can fire two rules.

Consider a very simple but strange PPS specification. Only one user event *e1* is defined and *x* is the only state variable. Initially the value of *x* is *a*. Rule 2's pre-condition is true and therefore subsumes rule 1.

Rule	user event	x
1	e1	a b
2	e1	 c

An SMV program using `ASSIGN` statements might look as follows.

```
MODULE main
VAR
    Event : { e1 };
    x : { a, b, c };

ASSIGN
```

```
    init(x) := a;
    next(x) :=
      case
        Event = e1 & x = a : b;
        Event = e1         : c;
        1                  : x;
      esac;
```

We expect that there exists a next state in which *x* is *b* and also a next state in which *x* is *c*. However, the verification results tell us the latter is not true. This is because the case expression textually imposes a priority order on the conditions.

```
SPEC EX(x = b) -- SMV reports that this specification is true
SPEC EX(x = c) -- SMV reports that this specification is false
```

Alternatively, we may use the `TRANS` and `INIT` statements. It is possible in SMV to specify the transition relation directly as a propositional formula in terms of the current and next values of the state variables. Any current/next state pairs is in the transition relation iff they make the formula true. Similarly, it is possible to give the set of possible initial states as a formula in terms of only current state variables. An SMV program modeling the same example using this alternative shows that both specifications are true.

```
MODULE main
VAR
    Event : { e1 };
    x : { a, b, c };

INIT x = a
TRANS (Event = e1 & x = a & next(x) = b) |
      (Event = e1         & next(x) = c) |
      ( !((Event = e1 & x = a) | (Event = e1)) & next(x) = x)

SPEC EX(x = b) -- SMV reports that this specification is true
SPEC EX(x = c) -- SMV reports that this specification is true
```

The SMV literature does not recommend the use of `TRANS` and `INIT` since it is possible to write logical absurdities using these features [4]. For example, one could specify the logical constant `0` to represent the transition relation, resulting in a system with no transitions at all. However, the above example justifies our need to use these features. Because they are dangerous, it is the responsibility of those writing translators to ensure appropriate use of `TRANS` and `INIT` statements. We will fulfill this responsibility in Section 5.5.

5.3. Deadlock

A CTL computation tree cannot have finite traces. That is, given a current state, there must exist at least one successor. However, a PPS specification might have a deadlock. This means that at some point none of the rules are enabled, and hence none of the user events can be selected.

We resolve this situation by defining a special “stuck” event. When a PPS dialog gets into a deadlock, the user can only take the stuck event and remains stuck forever. From the CTL machine’s point of view, there is still a “null” transition when the deadlock happens. The logical condition for deadlock is the complement of the disjunction of all rule’s pre-conditions.

Consider a very simple example as follows. Two user events *e1* and *e2* are defined and *x* is the only state variable. Initially the value of *x* is *a*. Therefore only event *e1* can be taken and *x* becomes *b*. Since the current state is *b*, only event *e2* can be taken and *x* becomes *c*. Now the current state is *c* and neither of the rules are enabled. So the dialog gets into a deadlock.

Rule	user event	x
1	e1	a b
2	e2	b c

An SMV translation is as follows. In addition to the normal user events, a special stuck event is included in the event declaration. For conciseness, we use `DEFINE` statements, which are analogous to macro definitions in an ordinary programming language. Two macros are defined for each rule: one corresponds to the pre-condition vector; the other corresponds to the post-condition vector. `no_enabled` denotes the situation where none of the rules' pre-condition are satisfied. `default` denotes that the dialog state does not change at all (when it is stuck).

```
MODULE main
VAR
  x: {a, b, c};
  Event: {e1, e2, Stuck};

DEFINE
   pre1 := x = a;
   pre2 := x = b;
   no_enabled := ! (pre1 | pre2);
   post1 := next(x) = b;
   post2 := next(x) = c;
   default := next(x) = x;

INIT -- assign initial states
     (x = a) &
     -- restrict user events to only those which are enabled
     -- if no user event is enabled, then the system gets stuck
     ((pre1 & Event = e1) |
      (pre2 & Event = e2) |
      (no_enabled & Event = Stuck))

TRANS -- execute transitions specified by the rules
      ((Event = e1 & pre1 & post1) |
       (Event = e2 & pre2 & post2) |
       (Event = Stuck & no_enabled & default)) &
      -- restrict user events to only those which are enabled
      -- if no user event is enabled, then the system gets stuck
      ((next(pre1) & next(Event) = e1) |
       (next(pre2) & next(Event) = e2) |
       (next(no_enabled) & next(Event) = Stuck))
```

Notice that in the `INIT` statement, the first part is to assign the initial states and the second part is to decide the enabled user events. In the `TRANS` statement, the first part is to assign values according to the post-conditions and the second part is to decide the enabled user events in the next state. Notice that `next(preN)` means that the next state satisfies the pre-condition of rule N.

5.4. Summary of Translation Steps

1. Declare the necessary state variables of the PPS dialog model.
2. Declare the user events of the PPS dialog model plus a special stuck event.
3. For each rule, define a macro to capture the pre-condition vector.
4. Define a macro which is the negation of the disjunction of all pre-conditions.
5. For each rule, define a macro to capture the post-condition vector.
6. Define a macro which specifies that the state of the dialog model does not change.
7. Write down the `INIT` statement which is a conjunction of two parts. The first part specifies the starting states. The second part specifies which user events are initially enabled.
8. Write down the `TRANS` statement which is a conjunction of two parts. The first part specifies the next state of the dialog model according to the post-conditions. The second part specifies which user events are enabled in the next state.

The above steps are a mechanical translation process and can be easily programmed.

5.5. Justifications

Formally, we are constructing a CTL machine which models the $PPS^{\dagger}$ specification. $PPS^{\dagger}$ is derived from *PPS* augmented by one additional rule to handle the deadlock. Suppose a_{N+1} is the special “stuck” event. $PPS^{\dagger} = (A^{\dagger}, \Sigma, T^{\dagger})$ where

- $A^{\dagger}$ is the set of user events plus the stuck event; that is, $A^{\dagger} = A \cup \{ a_{N+1} \}$, where $a_{N+1} \notin A$.
- Σ is the set of dialog states as defined in the original *PPS*.
- $T^{\dagger}$ is a binary relation where $T^{\dagger} \subseteq A^{\dagger} \times (\Sigma \rightarrow \Sigma)$, which additionally includes one rule to deal with the deadlock; that is, $T^{\dagger} = T \cup \{ (a_{N+1}, trans_{N+1}) \}$, where $pre_{N+1} = \Sigma - \bigcup_{1 \le i \le N} pre_i = \text{dom } trans_{N+1}$, and $trans_{N+1}$ is an identity function of Σ.

The domain of $trans_{N+1}$ is the complement of the union of all other rules’ pre-conditions. It maps any state vector to the same vector.

The translated SMV program essentially builds a machine of $CTL = (S, R, P)$ where

- S is the set of states each of which combines a user event with a dialog state ($S \subseteq A^{\dagger} \times \Sigma$).
- R is a binary relation where $R \subseteq (A^{\dagger} \times \Sigma) \times (A^{\dagger} \times \Sigma)$ which gives the possible transitions. Observing the logic formula specified in the `TRANS` statement, there are essentially two pieces: the first piece specifies the transition of the dialog state according to the rules while does not specifies the value of the next event; the second piece specifies the value of the next event according to the value of the next dialog state while ignores the values of the current state. Thus literally translating the logic formula into a set notation, we obtain

$$R = \bigcup_{1 \le i \le N+1} \{ ((a_i, s), (a', s')) \mid (s, s') \in trans_i \wedge a' \in A^{\dagger} \}$$
$$\cap$$
$$\bigcup_{1 \le j \le N+1} \{ ((a, s), (a_j, s')) \mid a \in A^{\dagger} \wedge s \in \Sigma \wedge s' \in \mathrm{dom}\ trans_j \}$$

- $P: S \rightarrow 2^{AP}$ assigns to each state the set of atomic propositions true in that state.

To justify that CTL is a valid translation of $PPS^{\dagger}$, we need to argue that for given user events e and e' and dialog states v and v', the following holds:

$$((e, v), (e', v')) \in R \Leftrightarrow (v, v') \in \bigcup T^{\dagger}.e \wedge v' \in \mathrm{dom}\ (\bigcup T^{\dagger}.e')^{1}.$$

Proof:

$$\begin{aligned} \bigcup T^{\dagger}.e &= \bigcup \{ trans_i \mid a_i = e,\ 1 \le i \le N+1 \} \\ &= \{ (s, s') \mid (s, s') \in trans_i \wedge a_i = e,\ 1 \le i \le N+1 \}. \end{aligned}$$

$$\begin{aligned} \bigcup T^{\dagger}.e' &= \bigcup \{ trans_j \mid a_j = e',\ 1 \le j \le N+1 \} \\ &= \{ (s', s'') \mid (s', s'') \in trans_j \wedge a_j = e',\ 1 \le j \le N+1 \}. \end{aligned}$$

$$\mathrm{dom}\ (\bigcup T^{\dagger}.e') = \{ s' \mid s' \in \mathrm{dom}\ trans_j \wedge a_j = e',\ 1 \le j \le N+1 \}.$$

$$\begin{aligned} & ((e, v), (e', v')) \in R \\ \Leftrightarrow\ & ((e, v), (e', v')) \in \bigcup_{1 \le i \le N+1} \{ ((a_i, s), (a', s')) \mid (s, s') \in trans_i \wedge a' \in A^{\dagger} \} \\ & \qquad \cap \\ & \qquad \bigcup_{1 \le j \le N+1} \{ ((a, s), (a_j, s')) \mid a \in A^{\dagger} \wedge s \in \Sigma \wedge s' \in \mathrm{dom}\ trans_j \} \\ \Leftrightarrow\ & ((e, v), (e', v')) \in \bigcup_{1 \le i \le N+1} \{ ((a_i, s), (a', s')) \mid (s, s') \in trans_i \wedge a' \in A^{\dagger} \} \\ & \qquad \wedge \\ & ((e, v), (e', v')) \in \bigcup \{ ((a, s), (a_j, s')) \mid a \in A^{\dagger} \wedge s \in \Sigma \wedge s' \in \mathrm{dom}\ trans_j \} \end{aligned}$$

---

1. The notation $R.s$ denotes the set of all the secondary values of pairs which correspond to s in a binary relation R; that is, $R.s = \{ t \mid (s, t) \in R \}$.

$1 \le j \le N+1$

$$\Leftrightarrow (e, v, v') \in \bigcup_{1 \le i \le N+1} \{ (a_i, s, s') \mid (s, s') \in trans_i \} \qquad \text{[for the argument of } \Leftarrow, e' \in A^{\dagger}\text{]}$$

$$\wedge$$

$$(e', v') \in \bigcup_{1 \le j \le N+1} \{ (a_j, s') \mid s' \in \mathrm{dom}\ trans_j \} \qquad \text{[for the argument of } \Leftarrow, e \in A^{\dagger} \text{ and } v \in \Sigma\text{]}$$

$$\Leftrightarrow (e, v, v') \in \bigcup_{1 \le i \le N+1 \wedge a_i = e} \{ (a_i, s, s') \mid (s, s') \in trans_i \}$$

$$\wedge$$

$$(e', v') \in \bigcup_{1 \le j \le N+1 \wedge a_j = e'} \{ (a_j, s') \mid s' \in \mathrm{dom}\ trans_j \}$$

$$\Leftrightarrow (e, v, v') \in \{ (a_i, s, s') \mid (s, s') \in trans_i \wedge a_i = e \wedge 1 \le i \le N+1 \}$$

$$\wedge$$

$$(e', v') \in \{ (a_j, s') \mid s' \in \mathrm{dom}\ trans_j \wedge a_j = e' \wedge 1 \le j \le N+1 \}$$

$$\Leftrightarrow (v, v') \in \{ (s, s') \mid (s, s') \in trans_i \wedge a_i = e \wedge 1 \le i \le N+1 \}$$

$$\wedge$$

$$v' \in \{ s' \mid s' \in \mathrm{dom}\ trans_j \wedge a_j = e' \wedge 1 \le j \le N+1 \}$$

$$\Leftrightarrow (v, v') \in \bigcup T^{\dagger}.e \wedge v' \in \mathrm{dom}\ (\bigcup T^{\dagger}.e'). \ \square$$

To justify that the transition relation in *CTL* has no finite traces, we need to argue that there always exists a successor for a reachable state; that is, we need to show the following:

$$((e, v), (e', v')) \in R \Rightarrow \exists\, e'' \in A^{\dagger}, v'' \in \Sigma \bullet ((e', v'), (e'', v'')) \in R.$$

Proof:

From the definition of R and the hypothesis of $((e, v), (e', v')) \in R$, we can infer that

$$\exists\, x.\ 1 \le x \le N+1 \bullet a_x = e' \wedge v' \in \mathrm{dom}\ trans_x.$$

Let $v'' = trans_x(v')$. We also know that $trans_x(v') \in \Sigma$ and $\Sigma = \bigcup_{1 \le i \le N+1} \mathrm{dom}\ trans_i$. Therefore, $v'' \in \bigcup_{1 \le i \le N+1} \mathrm{dom}\ trans_i$. This implies that

$$\exists\, y.\ 1 \le y \le N+1 \bullet v'' \in \mathrm{dom}\ trans_y.$$

Let $e'' = a_y$. Then $((e', v'), (e'', v'')) = ((a_x, v'), (a_y, v''))$. Given the definition of R,

$$((a_x, v'), (a_y, v'')) \in \bigcup_{1 \le i \le N+1} \{ ((a_i, s), (a', s')) \mid (s, s') \in trans_i \wedge a' \in A^{\dagger} \}$$

$$\cap$$

$$\bigcup_{1 \le j \le N+1} \{ ((a, s), (a_j, s')) \mid a \in A^{\dagger} \wedge s \in \Sigma \wedge s' \in \mathrm{dom}\ trans_j \}$$

$= R.$ ❒

6. Verification of Properties

Various properties can be analyzed against the PPS dialog in the translated CTL machine. In Section 6.1., we present a PPS dialog of a schedule organizer and use our translation technique to construct a SMV program for it. In Section 6.2., we identify several categories of questions and provide CTL templates for expressing these questions. In Section 6.3., we summarize a list of insights that we discovered in our attempt of establishing templates.

6.1. PPS Dialog of a Schedule Organizer

The example comes from a product of a well-known brand. The detailed features have been abstracted away and only the essential functionality is captured. A user can turn the organizer on/off, and the organizer is divided into several modes: schedule, calendar, telephone, memo,... etc. A user can access a mode, select a day, edit an appointment for a day, view a day's schedule, overview a month's schedule, delete an appointment, and perform some other actions like checking memo and searching a phone number.

We abstractly model the organizer with three modes: *Calendar*, *Schedule*, and *Other*. A user can switch the power *On* and *Off*. The field of *Today* records whether currently the system is at today. The field of *TargetDay* records whether currently the system is at a user's desired day. The field of *Editing* records whether a user is editing an appointment. The field of *Saved* records whether there is an appointment being edited without being saved. The PPS dialog is written in Table 2. Notice that the post-conditions of certain rules are intentionally specified in a non-deterministic way. For example, when the action of *SwitchOn* is taken, the system will lead to today as the current day but whether it is a user's desired day is the user's decision. Thus rules 1 and 2 are written for the *SwitchOn* action and the post-condition of *TargetDay* can be either *Yes* or *No*. Also notice that we put no limit on the number of appointments for a day. After selecting a desired day, a user can view the appointments for that day in sequence by continuously taking the *ViewNext* action. To signal that the appointments for a desired day have all been viewed, the post-condition of *TargetDay* is changed from *Yes* to *No*. However, this change is also a non-deterministic decision since the dialog model does not record how many appointments there are for each day. Four rules are written for the *ViewNext* action, because of the non-determinism, and because of the interaction between *Today* field and *TargetDay* field.

Fields Power: { On, Off }
Mode: { Schedule, Calendar, Other }
Today, TargetDay, Editing, Saved: { Yes, No }
Initial state : (Power=Off, Mode=Other, Editing=No, Saved=Yes)

Rule	user event	Power	Mode	Today	TargetDay	Editing	Saved
1	SwitchOn	Off On		 Yes	 Yes		

Rule	user event	Power	Mode	Today	TargetDay	Editing	Saved
2	SwitchOn	Off On		 Yes	 No		
3	SwitchOff	On Off				 No	
4	GetTodayS	On	 Schedule	 Yes	 Yes	 No	
5	GetTodayC	On	 Calendar	 Yes	 Yes	 No	
6	AccessSchedule	On	 Schedule			 No	
7	AccessCalendar	On	 Calendar			 No	
8	AccessOther	On	 Other	 Yes	 Yes	 No	
9	AccessOther	On	 Other	 Yes	 No	 No	
10	SpecifyDayS	On	Schedule	 Yes	 Yes	 No	
11	SpecifyDayS	On	Schedule	 No	 Yes	 No	
12	SpecifyDayC	On	Calendar	 Yes	 Yes	 No	
13	SepcifyDayC	On	Calendar	 No	 Yes	 No	
14	ViewNext	On	Schedule		Yes	 No	
15	ViewNext	On	Schedule	No Yes	Yes No	 No	
16	ViewNext	On	Schedule	No No	Yes No	 No	
17	ViewNext	On	Schedule	Yes No	Yes No	 No	
18	OverView	On	Calendar				
19	Edit	On	Schedule			 Yes	 No
20	Commit	On				Yes No	 Yes
21	DeleteOne	On	Schedule			No	
22	DelMonthAppt	On	Calendar			No	
23	DoOther	On	Other				

Table 2. PPS specification of a schedule organizer

By following the mechanical translation steps identified in Section 5.4., the SMV program of the schedule organizer can be easily obtained according to Table 2. The source program is included in Appendix I.

6.2. Category of Questions

In the following subsections, each explains one category of questions, provides a template for casting the questions as CTL formulae, and gives verification examples on the schedule organizer. Note that in the template description, *s-stmt* denotes a logic formula expressed in terms of a state value or a conjunction of state values (e.g., `Power=Off`, `Today=Yes & Editing=No`); *e-stmt* denotes a logic formula expressed in terms of an event label (e.g., `event=AccessSchedule`, `event=Deleteone`). If there is more than one *stmt* emerging in the template, the numeric suffix denotes whether the *stmt*'s should be the same or could be different.

6.2.1. Rule Set Connectedness

Question: Given initialization, can all rules somehow be enabled?

We assume that all rules are put forth for some purpose. If some rule is impossible to fire, there is potentially a problem in the dialog design. The designer needs to either revisit the dialog design or simply eliminate the useless rule. In the following template, *e-stmt* expresses the event associated with a rule, and *s-stmt* captures the pre-condition of that rule.

Template: `EF(`*e-stmt* `&` *s-stmt*`)` for each rule

Example:

```
EF(event=SwitchOn & Power=Off)
EF(event=SwitchOff & Power=On)
EF(event=GetTodayS & Power=On)
EF(event=GetTodayC & Power=On)
EF(event=AccessSchedule & Power=On)
EF(event=AccessCalendar & Power=On)
EF(event=AccessOther & Power=On)
EF(event=SpecifyDayS & Power=On & Mode=Schedule)
EF(event=SpecifyDayC & Power=On & Mode=Calendar)
EF(event=ViewNext & Power=On & Mode=Schedule & TargetDay=Yes)
EF(event=ViewNext & Power=On & Mode=Schedule & Today=No &
                   TargetDay=Yes)
EF(event=ViewNext & Power=On & Mode=Schedule & Today=Yes &
                   TargetDay=Yes)
EF(event=OverView & Power=On & Mode=Calendar)
EF(event=Edit & Power=On & Mode=Schedule)
EF(event=Commit & Power=On & Editing=Yes)
EF(event=DeleteOne & Power=On & Mode=Schedule & Editing=No)
EF(event=DelMonthAppt & Power=On & Mode=Calendar & Editing=No)
EF(event=DoOther & Power=On & Mode=Other)
```

The above formulae verify whether all rules in the schedule organizer dialog example can eventually be enabled. SMV reports that all are true.

6.2.2. Free of Deadlock

Question: Given initialization, is it true that the dialog will never get into a state where no user events can be taken?

That is, we want to avoid the situation where none of the regular user events are enabled, and the only event that a user can take is the special *"stuck"* event. Such a situation would of course be very annoying to a user.

Template: `!EF(event=stuck)` or equivalently, `!EF(no_enabled)`

Example:
```
!EF(event=stuck)
!EF(no_enabled)
```

Either of the above formulae is sufficient to verify this property, since the two formulae are equivalent in our translated CTL model. SMV reports both true.

6.2.3. Free of Live Deadlock

Question: From any state of a given state set, can the user escape from that state set?

In the template, *s-stmt1* expresses the state set to escape from.

Template: `AG(`*s-stmt1* `-> EF(!`*s-stmt1*`))`

Example:
```
AG(Mode=Other -> EF(!Mode=Other))
AG(Mode=Schedule -> EF(!Mode=Schedule))
AG(Mode=Calendar -> EF(!Mode=Calendar))
```

The above formulae verify if a user can always escape from the *Schedule* mode, *Calendar* mode, and *Other* mode, respectively. SMV reports all true.

6.2.4. Weak Task Completeness

Question: Can a user find some way to accomplish a task from initialization? Accomplishment of a task is represented in terms of a particular state that could be reached or a particular user action that could be taken.

In the template, *s-stmt* and *e-stmt* express the tasks need to be accomplished.

Template: `EF(`*s-stmt*`)` or `EF(`*e-stmt*`)`

Example:
```
EF(Today=Yes)
EF(TargetDay=Yes & Editing=Yes & Saved=No)
EF(event=OverView)
EF(Mode=Calendar & Editing=Yes)
```

The first formula checks if the system can reach a state where today is the current day. The second formula checks if the system can reach a state where a user can specify a desired day and edit an appointment while the appointment is not yet saved. The third formula checks if a user can overview a month's schedule. The fourth formula checks if a user can edit something while in the *Calendar* mode. SMV reports that the first three formulae are true but the fourth one is false. It is obvious that the last one does not hold since the system must be in the *Schedule* mode before a user can edit an appointment.

6.2.5. Strong Task Completeness

Question: Can the dialog model inevitably lead a user to accomplish an important task from initialization? Accomplishment of a task is also represented in terms of a particular state that could be reached or a particular user action that could be taken.

This property is valuable for a novice user. The dialog inevitably leads the user to try some important or most frequently used functions without the user searching which actions to take. On the other hand, we might want a negative answer to this question if we want to avoid a novice user unintentionally performing some undesired tasks such as changing the time setting of a watch when viewing the time display. In the template, *s-stmt* and *e-stmt* express the tasks need to be accomplished.

Template: `AF(`*s-stmt*`)` or `AF(`*e-stmt*`)`

Example:
```
AF(Today=Yes)
AF(TargetDay=Yes & Editing=Yes & Saved=No)
AF(event=OverView)
AF(Mode=Calendar & Editing=Yes)
```

The formulae check if the four tasks identified in the previous subsection can be inevitably completed. Only the first one is true because initially only *SwitchOn* action is enabled and *SwitchOn* leads to today as the current day.

6.2.6. State Inevitability

Question: From any state of a given state set, will the dialog model absolutely take the user to a critical state?

We usually want to make sure no matter how the user navigates through the system, the user can absolutely get to an important state or a frequently needed state. In the template, *s-stmt1* is the given state set, and *s-stmt2* is the target state set.

Template: `AG(`*s-stmt1* `-> AF(`*s-stmt2*`))`

Example:
```
AG(Mode=Other -> AF(TargetDay=Yes))
AG(Mode=Other -> AF(Today=Yes))
AG(Mode=Other -> AF(Mode=Schedule & Today=Yes))
AG(Editing=Yes & Saved=No -> AF(Saved=Yes))
```

The above four formulae should be self-explanatory. SMV reports only the second one is true. However, it is highly desired that the fourth formula holds, which ensures that a user will never accidentally lose what he is editing. This desired property would hold if we change the rule set in a way that a user must “commit” the edited appointment before leaving editing.

6.2.7. Weak Task Connectedness

Question: From any state of a given state set, no matter which enabled action a user is going to take, can the user find some way to get to a target state set?

In the template, *s-stmt1* is the given state set, and *s-stmt2* is the target state set.

Template: `AG(`*s-stmt1* `-> EF(`*s-stmt2*`))`

Example:
```
AG(Today=No & TargetDay=No -> EF(Today=Yes & TargetDay=Yes))
AG(TargetDay=Yes & Editing=Yes & Saved=No -> EF(Saved=Yes))
AG(Mode=Calendar & Editing=Yes -> EF(Mode=Schedule))
AG(Mode=Schedule -> EF(Mode=Calendar & Editing=Yes))
```

The first formula checks whether a user can get to today as the desired day if currently the system is not at today and is not at a user's desired day. The second formula checks whether a user can save an edited appointment if currently the system is at a desired day, a user is editing appointment, and the appointment is not yet saved. The first two formulae are verifiably true. Notice that the third formula is trivially true because the dialog will never reach a state where the system is in the *Calendar* mode and at the same time a user is editing an appointment. Also notice that the fourth formula is false because of the same reason.

It is important to notice that this template generally poses a question which is stronger than needed. Since we include the event to be taken as part of the state information, if the property of *AG(s=s1 -> EF(s=s2))* holds, it not only guarantees that there exists some way to arrive at *s2* from *s1*, but also means that "whatever enabled action a user is going to take when the system is in the state set of *s1*, there exists a way to arrive at *s2*." Alternatively, we might provide a template as follows:

Template: `EF(`*s-stmt1* `& EF(`*s-stmt2*`))`

However, this template poses a question which is too weak. If the property of *EF(s=s1 & EF(s=s2))* holds, it only means that there exists a particular state, say *t,* in the state set of *s1* (where $t \in s1$), and from *t* there exists a way to arrive at *s2*.

There is one way to express exactly the question we want to ask, as illustrated in the following template. This template, however, requires an exhaustive enumeration of all possible events.

Template: `AG((`*s-stmt1* `&` *e-stmt1* `-> EF(`*s-stmt2*`)) |`
`(`*s-stmt1* `&` *e-stmt2* `-> EF(`*s-stmt2*`)) |`
`(`*s-stmt1* `&` *e-stmt3* `-> EF(`*s-stmt2*`)) |`
. . . for each event in the event set`)`

We choose to provide a stronger template because it subsumes what is generally asked and does not need to explicitly list all elements in the event set.

6.2.8. Strong Task Connectedness

Question: From any state of a given state set, no matter which enabled action a user is going to take, can the user find some way to get to a target state set via a particular user action? This particular user action is supposed to be the last action.

This property makes sense since a user may easily remember one particular user action to accomplish a desired task. In the template, *s-stmt1* is the given state set, *e-stmt* is the particular user event, and *s-stmt2* is the target state set. Notice that the template also poses a question which is stronger than the one generally asked because it considers whatever enabled action a user initially is going to take.

Template: `AG(`*s-stmt1* `-> EF(`*e-stmt* `& AX(`*s-stmt2*`)))`

Example:
```
AG(Today=No & TargetDay=No
   -> EF(event=GetTodayS & AX(Today=Yes & TargetDay=Yes)))
AG(Today=No & TargetDay=No
   -> EF(event=SpecifyDayC & AX(Today=Yes & TargetDay=Yes)))
```

The first formula checks whether, by doing *GetTodayS*, a user can get to today as the desired day if currently the system is not at today and is not at a user's desired day. This is verifiably true. The second formula checks whether a user can achieve the same by doing *SpecifyDayC*, and this is false. This result does not matter since we intentionally non-deterministically define *Today*'s post-condition to be either *Yes* or *No*.

6.2.9. Rule Reversibility

Question: Given a rule which is going to fire, is it possible to eventually reverse the state back to the original situation after the rule fires?

A user may want to reverse the effect of a rule in terms of getting back to the same choice of user events. In the template, *e-stmt* expresses the event associated with a rule, and *s-stmt1* captures the pre-condition of that rule.

Template: `AG(`*e-stmt* `&` *s-stmt1* `-> EX EF(`*s-stmt1*`))`

Example:
```
AG(event=ViewNext &
   Power=On & Mode=Schedule & Today=No & TargetDay=Yes
  -> EX EF(Power=On & Mode=Schedule & Today=No & TargetDay=Yes))
```

The above example verifies this property for rule 15 and rule 16 as they have the same pre-condition, and this formula verifiably holds.

6.2.10. Undo within N Steps

Question: From any state of a given state set, if the next state leads the system out of the state set, can a user go back to the given state set within N steps?

Template: `AG(`*s-stmt1* `-> EX(!`*s-stmt1* `-> (EX(`*s-stmt1*`) | EX EX(`*s-stmt1*`) |` ... N times`))`

Example:
```
AG(Editing=Yes -> EX(!Editing=Yes -> EX(Editing=Yes)))
AG(Editing=Yes -> EX(!Editing=Yes -> EX(Editing=Yes) |
                                     EX EX(Editing=Yes)))
```

The first formula verifies that once a user leaves editing he can go back to edit within one step, and the tool reports it is false. The second formula verifies the same within two steps, and the tool reports it is true.

It is critical to note that the template is valid only for state variables whose post-conditions are deterministically decided. Suppose that for a particular action the variable *s* can be non-deterministically assigned as either *s1* or *s2* and the property of *AG(s=s1 -> EX(!s=s1 -> EX(s=s1)))* is true. This does not guarantee that *s1* can be undone within one step because *s* may be assigned as *s1* and this makes *!s=s1 -> EX(s=s1)* vacuously true and therefore *EX(!s=s1 -> EX(s=s1))* true. However, if there exists a case in which *s1* cannot be undone within one step when *s* is assigned as *s2*, then the verification result gives a fake conclusion.

6.2.11. Accessibility

Question: From any reachable state, can the user find some way to get to some critical state set (such as the help system)?

In the template, *s-stmt* expresses the critical state set.

Template: `AG EF(`*s-stmt*`)`

Example: `AG EF(Today=Yes & TargetDay=Yes)`

To use a schedule organizer, it is frequently necessary to get to today as the desired day at any time. The example verifies this property and the result is true.

6.2.12. Accessibility within N Steps

Question: From any reachable state, no matter which enabled action a user is going to take, can the user find some way to get to some critical state set within N steps (such as the help system)?

In the template, *s-stmt1* expresses the critical state set. Notice that since we include event label as a state variable, checking accessibility within one step usually ends up with falseness. For example, a simple counterexample against the property of *AG(EX(state=s))* would be a state different from *s* with some user event *e* (which is going to be taken) which retains the state. This kind of checking makes more sense if the property is asked for *N > 1*.

Template: `AG(EX(`*s-stmt1*`) | EX EX(`*s-stmt1*`) |` ... N times`)`

Example:
```
AG(EX(Today=Yes & TargetDay=Yes))
AG(EX(Today=Yes & TargetDay=Yes) |
   EX EX(Today=Yes & TargetDay=Yes))
```

The first formula checks if today can be designated as the desired day within one step and the result is false. A counterexample is when the system is at the state of *Today=No* and *TargetDay=No*, and *event=AccessSchedule*, there exists no next state in which *Today=Yes* and *TargetDay=Yes*, since the action of *AccessSchedule* will not change the variables of *Today* and *TargetDay*. The second formula checks if *Today=Yes* and *TargetDay=Yes* can be accessed within two steps from anywhere and the result is true because the dialog provides the *GetTodayS* and *GetTodayC* actions.

6.2.13. State Avoidability

Question: From a given state set, can a user reach a target state set without entering an undesired state set?

The reason for this kind of checking is that sometimes it is annoying to inevitably get into an undesired state. In the template, *s-stmt1* is the given state set, *s-stmt2* is the undesired state set, and *s-stmt3* is the target state set, and *e-stmt1*, *e-stmt2*,... etc. are the events that the user will not perform.

Template: `AG(`*s-stmt1* `& !`*s-stmt2* `& !`*e-stmt1* `& !`*e-stmt2* `&` ... as necessary
`-> E[!(`*s-stmt2*`) U (`*s-stmt3*`)])`

Example:
```
AG(Mode=Schedule &
   !Editing=Yes &
   !event=Edit
   -> E[!Editing=Yes U Mode=Calendar])
AG(Power=On &
   !Today=Yes &
   !event=GetTodayS & !event=GetTodayC
   -> E[!Today=Yes U TargetDay=Yes])
AG(Mode=Calendar &
   !TargetDay=Yes &
   !event=SpecifyDayC & !event=SpecifyDayS &
   !event=GetTodayS & !event=GetTodayC
   -> E[!TargetDay=Yes U Editing=Yes])
```

All three formulae hold. The first formula checks that, from the *Schedule* mode, a user can get to the *Calendar* mode without the need of editing an appointment. During this course, certainly the user will not choose to edit an appointment as otherwise it is the user who intentionally edits something. The second formula checks that from any state when the power is on, a user can specify a desired day without having today as the current day. The third formula checks that from the *Calendar* mode, a user can edit an appointment without designating a specific day.

It is interesting to notice that, because our CTL machine includes the event that a user can take as part of the state information, the process of checking this property may require incrementally filtering out the events which a user will not select to perform. For example, to check the first case, the designer might initially specify something like

```
AG(Mode=Schedule &
   !Editing=Yes &
   -> E[!Editing=Yes U Mode=Calendar])
```

This would result in falseness since the formula did not eliminate the *Edit* action. By looking through the counterexample, the designer can decide whether the event that the counterexample provides makes sense when a user wants to avoid the state of *Editing=Yes*. Then the designer can filter out the *Edit* event that a user will not perform. However, in some cases it is clumsy to filter out the events one by one. For example, the last case requires explicitly filtering out four user events.

6.2.14. Event Constraint

Question: Does the dialog model ensure/prohibit a particular user action for a given state set?

In the template, *s-stmt* expresses the given state set, and *e-stmt* denotes the user event which needs to be assured/prohibited.

Template: `AG(`*s-stmt* `->` *e-stmt*`)` or `AG(`*s-stmt* `-> !`*e-stmt*`)`

Example: `AG(Editing=Yes -> !event=DeleteOne)`

If a user can delete an appointment when editing an appointment, it would be confusing to the user since which appointment will be deleted is not clear. The example checks whether a user is disallowed to perform deletion when editing. This property holds for the dialog.

6.2.15. Feature Assurance

Question: Does the dialog model assure a desired feature in a given state set?

In the template, *s-stmt1* expresses the given state set and *s-stmt2* the desired feature.

Template: `AG(`*s-stmt1* `->` *s-stmt2*`)`

Example: `AG(Editing=No -> Saved=Yes)`

The example checks if there is no appointment lost when a user is not editing. This property is false for this dialog since when a user is editing, there are lots of ways to leave editing without saving the edited appointment first. This suggests a way to modify the dialog as Section 6.2.6. does.

6.3. Summary of Insights

Firstly, in our attempt of providing templates for frequently asked questions, some difficulties arise because we include the event (that a user can take) as part of the state information in our translated CTL machine. This results in the following undesired facts:

- The templates given in Section 6.2.7. and Section 6.2.8. are stronger than needed.
- The template given in Section 6.2.12. generally does not make sense if the question is asked for $N = 1$.

- The template given in Section 6.2.13. needs to explicitly filter out certain events that a user will not select to take.

To eliminate this problem, one possibility is to use the approach by Atlee and Gannon [9]. They model the event-based system using two distinct kinds of states: an *EXIT* state determines which event to take, a *~EXIT* state captures the result of firing an event. The system runs alternately between an *EXIT* state and a *~EXIT* state. However, their translation is based on a different kind of requirements specification called SCR, which is more complex than PPS. At this point, we are unable to say whether their approach can resolve our difficulties identified above.

Secondly, we allow non-determinism in a PPS dialog description because we believe it makes a PPS description more expressive when the dialog design is conducted at a higher level (e.g., we put no limit on the number of appointments for each day in the scheduler organizer example). However, non-determinism inhibits the designer from using the template given in Section 6.2.10. This leaves a trade-off for the designer: either constructing a deterministic PPS specification (less expressive) and being able to check the undo property, or writing a non-deterministic one (more expressive) and unable to check the undo property.

Thirdly, to question the accessibility and reversibility properties, the designer usually wants to know directly how easy it is to access a critical state or to undo an effect. However, the templates we provide can only check *whether* something can *eventually* be done, or something can be done *within certain steps*. It would be helpful if the tool can compute the maximum and minimum steps needed to access a particular state or to undo an undesired effect. The new advances in model checking show there are algorithms to accomplish this and have already been incorporated in the tool [10]. This potentially can provide the ability of computing the "cost functions" for reversibility and accessibility.

7. Animated Feedback for Fix-up and Improvement

If SMV tool reports that a CTL formula is false, it will try to give a counterexample output in a log file. The counterexample includes a series of state transitions starting from an initial state which violates the property being checked. Note that in the printout of an execution sequence by SMV, only the values of variables that change are printed. We are particularly interested in the *event* variable since its change suggests the sequence of user actions that explains why and how the property does not hold. In addition, the variable of pre_i ($1 \leq i \leq N$) indicates whether rule i is enabled.

Currently our approach of feeding the counterexample back to the animation tool is still primitive. We examine the values of *event* variable and manually perform the sequence of user events in the Action Simulator. For example, the counterexample for the property of *AG(Editing=Yes & Saved=No -> AF(Saved=Yes))* is listed in Appendix II. This is a desired property since it ensures that an unsaved appointment needs to be inevitably saved and thus a user will never accidentally lose an edited appointment. The counterexample suggests a sequence of actions: *SwitchOn*, *AccessSchedule*, *Edit*, *AccessSchedule*, *Edit*, *AccessSched-*

ule,... (repeat *Edit* and *AccessSchedule* forever). From the experience of executing this sequence in Action Simulator, the problem is that there are lots of actions a user can choose when the user is editing something. Only the *Commit* action will actually save the appointment while the user may accidentally choose one of the other enabled actions (such as *SwitchOff*, *AccessSchedule*, *AccessCalendar*, *AccessOther*, *GetTodayS*, *GetTodayC*,... etc.) without saving the edited appointment. One way to refine this situation is to have *Editing=No* being a pre-condition of all actions except *Edit* and *Commit*, and therefore a user must commit the appointment before leaving editing.

There are several problems of this primitive approach. First, the SMV tool will not give an indication in the case where it cannot generate a counterexample. The tool simply prints out an initial state and nothing else. When only an initial state is printed out, we need to figure out whether it is a "real" counterexample or it implies that a counterexample cannot be provided. Second, we need to locate at which point(s) of the counterexample the property being checked is violated. This means that we must, based on the property being verified, understand the "context" of scenario given by the counterexample and interpret the counterexample appropriately. Third, since we allow multiple rules to be associated with the same event label (as we want to allow non-determinism), we cannot decide exactly which rule to fire by only examining the *event* variable in the counterexample. We also need to examine the other state variables, especially the ones which are non-deterministically defined, to properly select which rule to fire. At this point, we have not yet resolved these issues and cannot provide an automated process for animated feedback.

8. Conclusion and Future Work

Our work demonstrates that it is possible to link an event-based tabular dialog specification with the powerful model checking technology, whereby a PPS dialog can be modeled as a CTL state-based machine and analyzed using the SMV model checker. The result is a method whose specification interface is intuitive and whose analysis is automatable. The mechanical translation from PPS to SMV we present in this report means that this process can be automated. We also provide a collection of CTL templates to answer certain kinds of frequently asked questions. This can reduce the need of learning deep model checking techniques for average programmers. We also show how a counterexample of a property can be used to improve the dialog design. However, there are many directions that this research can be carried on.

We have already built the automatable translation process from PPS to SMV. It is desired to have a corresponding automatable process to tie the knot from SMV back to PPS through Action Simulator or some other appropriate simulation tool. This can give more sensible feedback to the designer via visualization and therefore provide better facilities to improve the design. Moreover, the collection of templates presented in this report is crude and simply a start. It would be extremely helpful to have a codification of templates available to ordinary designers. The goal is to build a tool to automate step 3 and step 5 identified in Figure 1, and incorporate the codification into it.

Some difficulties of establishing templates resulted from the fact that our translated CTL machine involves the enabled user events as state information. Atlee and Gannon used a two-stage transition approach to model a different form of event-driven specifications [9]. An interesting subject is to investigate if their approach can better support our needs.

Properties concerning accessibility and reversibility generally require to know how easy a state can be accessed or how easy an effect can be undone. Campos, Clarke, et al. presented algorithms to calculate the minimum and maximum lengths over paths leading from a set of starting states to a set of final states [10]. We believe that this advance can readily be adopted by our approach to compute the cost functions of many properties.

The relationship of our work to Statemate system [13] needs to be examined. There are at least two interesting topics in this direction. First, it would be useful to know the difference of the kinds of properties analyzable by our approach and Statemate. Second, our tabular specification grows flat as we add rules. However, a system usually has different aspects of behavior which are orthogonal with each other, and the designer should be able to focus on one aspect at a time. The statechart [11][12], the foundation of Statemate, provides a hierarchical way to specify complex state transitions. We need to investigate if it is possible to impose some kind of structure on our tabular specification.

Appendix I

```
MODULE main
VAR
  Power     : { Off, On };
  Mode      : { Schedule, Calendar, Other };
  Today     : { Yes, No };
  TargetDay : { Yes, No };
  Editing   : { Yes, No };
  Saved     : { Yes, No };
  event     : { SwitchOn, SwitchOff, GetTodayS, GetTodayC, AccessSchedule,
                AccessCalendar, AccessOther, SpecifyDayS, SpecifyDayC,
                ViewNext, OverView, Edit, Commit, DeleteOne, DelMonthAppt,
                DoOther, stuck };

DEFINE
  pre1 := Power=Off;
  pre2 := Power=Off;
  pre3 := Power=On;
  pre4 := Power=On;
  pre5 := Power=On;
  pre6 := Power=On;
  pre7 := Power=On;
  pre8 := Power=On;
  pre9 := Power=On;
  pre10 := Power=On & Mode=Schedule;
  pre11 := Power=On & Mode=Schedule;
  pre12 := Power=On & Mode=Calendar;
  pre13 := Power=On & Mode=Calendar;
  pre14 := Power=On & Mode=Schedule & TargetDay=Yes;
  pre15 := Power=On & Mode=Schedule & Today=No & TargetDay=Yes;
  pre16 := Power=On & Mode=Schedule & Today=No & TargetDay=Yes;
```

```
  pre17 := Power=On & Mode=Schedule & Today=Yes & TargetDay=Yes;
  pre18 := Power=On & Mode=Calendar;
  pre19 := Power=On & Mode=Schedule;
  pre20 := Power=On & Editing=Yes;
  pre21 := Power=On & Mode=Schedule & Editing=No;
  pre22 := Power=On & Mode=Calendar & Editing=No;
  pre23 := Power=On & Mode=Other;
  no_enabled := ! (pre1 | pre2 | pre3 | pre4 | pre5 | pre6 | pre7 | pre8 |
                  pre9 | pre10 | pre11 | pre12 | pre13 | pre14 | pre15 | pre16 |
                  pre17 | pre18 | pre19 | pre20 | pre21 | pre22 | pre23);
  post1 := next(Power)=On & next(Mode)=Mode & next(Today)=Yes &
           next(TargetDay)=Yes & next(Editing)=Editing & next(Saved)=Saved;
  post2 := next(Power)=On & next(Mode)=Mode & next(Today)=Yes &
           next(TargetDay)=No & next(Editing)=Editing & next(Saved)=Saved;
  post3 := next(Power)=Off & next(Mode)=Mode & next(Today)=Today &
           next(TargetDay)=TargetDay & next(Editing)=No & next(Saved)=Saved;
  post4 := next(Power)=Power & next(Mode)=Schedule & next(Today)=Yes &
           next(TargetDay)=Yes & next(Editing)=No & next(Saved)=Saved;
  post5 := next(Power)=Power & next(Mode)=Calendar & next(Today)=Yes &
           next(TargetDay)=Yes & next(Editing)=No & next(Saved)=Saved;
  post6 := next(Power)=Power & next(Mode)=Schedule & next(Today)=Today &
           next(TargetDay)=TargetDay & next(Editing)=No & next(Saved)=Saved;
  post7 := next(Power)=Power & next(Mode)=Calendar & next(Today)=Today &
           next(TargetDay)=TargetDay & next(Editing)=No & next(Saved)=Saved;
  post8 := next(Power)=Power & next(Mode)=Other & next(Today)=Yes &
           next(TargetDay)=Yes & next(Editing)=No & next(Saved)=Saved;
  post9 := next(Power)=Power & next(Mode)=Other & next(Today)=Yes &
           next(TargetDay)=No & next(Editing)=No & next(Saved)=Saved;
  post10 := next(Power)=Power & next(Mode)=Mode & next(Today)=Yes &
            next(TargetDay)=Yes & next(Editing)=No & next(Saved)=Saved;
  post11 := next(Power)=Power & next(Mode)=Mode & next(Today)=No &
            next(TargetDay)=Yes & next(Editing)=No & next(Saved)=Saved;
  post12 := next(Power)=Power & next(Mode)=Mode & next(Today)=Yes &
            next(TargetDay)=Yes & next(Editing)=No & next(Saved)=Saved;
  post13 := next(Power)=Power & next(Mode)=Mode & next(Today)=No &
            next(TargetDay)=Yes & next(Editing)=No & next(Saved)=Saved;
  post14 := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
            next(TargetDay)=TargetDay & next(Editing)=No & next(Saved)=Saved;
  post15 := next(Power)=Power & next(Mode)=Mode & next(Today)=Yes &
            next(TargetDay)=No & next(Editing)=No & next(Saved)=Saved;
  post16 := next(Power)=Power & next(Mode)=Mode & next(Today)=No &
            next(TargetDay)=No & next(Editing)=No & next(Saved)=Saved;
  post17 := next(Power)=Power & next(Mode)=Mode & next(Today)=No &
            next(TargetDay)=No & next(Editing)=No & next(Saved)=Saved;
  post18 := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
            next(TargetDay)=TargetDay & next(Editing)=Editing & next(Saved)=Saved;
  post19 := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
            next(TargetDay)=TargetDay & next(Editing)=Yes & next(Saved)=No;
  post20 := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
            next(TargetDay)=TargetDay & next(Editing)=No & next(Saved)=Yes;
  post21 := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
            next(TargetDay)=TargetDay & next(Editing)=Editing & next(Saved)=Saved;
  post22 := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
            next(TargetDay)=TargetDay & next(Editing)=Editing & next(Saved)=Saved;
  post23 := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
            next(TargetDay)=TargetDay & next(Editing)=Editing & next(Saved)=Saved;
  default := next(Power)=Power & next(Mode)=Mode & next(Today)=Today &
             next(TargetDay)=TargetDay & next(Editing)=Editing & next(Saved)=Saved;

INIT
  (Power=Off & Mode=Other & Editing=No & Saved=Yes) &
  ((pre1 & event=SwitchOn) |
   (pre2 & event=SwitchOn) |
   (pre3 & event=SwitchOff) |
   (pre4 & event=GetTodayS) |
```

```
   (pre5 & event=GetTodayC) |
   (pre6 & event=AccessSchedule) |
   (pre7 & event=AccessCalendar) |
   (pre8 & event=AccessOther) |
   (pre9 & event=AccessOther) |
   (pre10 & event=SpecifyDayS) |
   (pre11 & event=SpecifyDayS) |
   (pre12 & event=SpecifyDayC) |
   (pre13 & event=SpecifyDayC) |
   (pre14 & event=ViewNext) |
   (pre15 & event=ViewNext) |
   (pre16 & event=ViewNext) |
   (pre17 & event=ViewNext) |
   (pre18 & event=OverView) |
   (pre19 & event=Edit) |
   (pre20 & event=Commit) |
   (pre21 & event=DeleteOne) |
   (pre22 & event=DelMonthAppt) |
   (pre23 & event=DoOther) |
   (no_enabled & event=stuck))

TRANS
   ((event=SwitchOn & pre1 & post1) |
    (event=SwitchOn & pre2 & post2) |
    (event=SwitchOff & pre3 & post3) |
    (event=GetTodayS & pre4 & post4) |
    (event=GetTodayC & pre5 & post5) |
    (event=AccessSchedule & pre6 & post6) |
    (event=AccessCalendar & pre7 & post7) |
    (event=AccessOther & pre8 & post8) |
    (event=AccessOther & pre9 & post9) |
    (event=SpecifyDayS & pre10 & post10) |
    (event=SpecifyDayS & pre11 & post11) |
    (event=SpecifyDayC & pre12 & post12) |
    (event=SpecifyDayC & pre13 & post13) |
    (event=ViewNext & pre14 & post14) |
    (event=ViewNext & pre15 & post15) |
    (event=ViewNext & pre16 & post16) |
    (event=ViewNext & pre17 & post17) |
    (event=OverView & pre18 & post18) |
    (event=Edit & pre19 & post19) |
    (event=Commit & pre20 & post20) |
    (event=DeleteOne & pre21 & post21) |
    (event=DelMonthAppt & pre22 & post22) |
    (event=DoOther & pre23 & post23) |
    (event=stuck & no_enabled & default)) &
   ((next(pre1) & next(event)=SwitchOn) |
    (next(pre2) & next(event)=SwitchOn) |
    (next(pre3) & next(event)=SwitchOff) |
    (next(pre4) & next(event)=GetTodayS) |
    (next(pre5) & next(event)=GetTodayC) |
    (next(pre6) & next(event)=AccessSchedule) |
    (next(pre7) & next(event)=AccessCalendar) |
    (next(pre8) & next(event)=AccessOther) |
    (next(pre9) & next(event)=AccessOther) |
    (next(pre10) & next(event)=SpecifyDayS) |
    (next(pre11) & next(event)=SpecifyDayS) |
    (next(pre12) & next(event)=SpecifyDayC) |
    (next(pre13) & next(event)=SpecifyDayC) |
    (next(pre14) & next(event)=ViewNext) |
    (next(pre15) & next(event)=ViewNext) |
    (next(pre16) & next(event)=ViewNext) |
    (next(pre17) & next(event)=ViewNext) |
    (next(pre18) & next(event)=OverView) |
    (next(pre19) & next(event)=Edit) |
```

```
    (next(pre20) & next(event)=Commit) |
    (next(pre21) & next(event)=DeleteOne) |
    (next(pre22) & next(event)=DelMonthAppt) |
    (next(pre23) & next(event)=DoOther) |
    (next(no_enabled) & next(event)=stuck))
```

Appendix II

```
-- specification AG (Editing = Yes & Saved = No -> AF Sav... is false
-- as demonstrated by the following execution sequence
state 11.1:
default = 0
post23 = 0
post22 = 0
post21 = 0
post20 = 0
post19 = 0
post18 = 0
post17 = 0
post16 = 0
post15 = 0
post14 = 0
post13 = 0
post12 = 0
post11 = 0
post10 = 0
post9 = 0
post8 = 0
post7 = 0
post6 = 0
post5 = 0
post4 = 0
post3 = 0
post2 = 0
post1 = 0
no_enabled = 0
pre23 = 0
pre22 = 0
pre21 = 0
pre20 = 0
pre19 = 0
pre18 = 0
pre17 = 0
pre16 = 0
pre15 = 0
pre14 = 0
pre13 = 0
pre12 = 0
pre11 = 0
pre10 = 0
pre9 = 0
pre8 = 0
pre7 = 0
pre6 = 0
pre5 = 0
pre4 = 0
pre3 = 0
pre2 = 1
pre1 = 1
Power = Off
Mode = Other
```

```
Today = No
TargetDay = No
Editing = No
Saved = Yes
event = SwitchOn

state 11.2:
pre23 = 1
pre9 = 1
pre8 = 1
pre7 = 1
pre6 = 1
pre5 = 1
pre4 = 1
pre3 = 1
pre2 = 0
pre1 = 0
Power = On
Today = Yes
event = AccessSchedule

state 11.3:
pre23 = 0
pre21 = 1
pre19 = 1
pre11 = 1
pre10 = 1
Mode = Schedule
event = Edit

-- loop starts here --
state 11.4:
pre21 = 0
pre20 = 1
Editing = Yes
Saved = No
event = AccessSchedule

state 11.5:
pre21 = 1
pre20 = 0
Editing = No
event = Edit

state 11.6:
pre21 = 0
pre20 = 1
Editing = Yes
event = AccessSchedule
```

References

[1] Dix, A., Finlay, J., Abowd, G. and Beale, R. *Human-Computer Interaction*. Prentice-Hall International, 1993.

[2] Gieskens, D. and Foley, J. Controlling user interface objects through pre- and post-conditions. *Human Factors in Computing Systems: Proceedings of CHI'92.* ACM Press, pp. 189-194, 1992.

[3] Monk, A. F. and Curry, M. B. Discount dialogue modelling with Action Simulator.

In *People and Computers IX: Proceedings of HCI'94*. Cambridge University Press, 1994 In press.

[4] McMillan, K. *Symbolic Model Checking: An approach to the state explosion problem*. Carnegie Mellon University, Computer Science Department, Technical Report CMU-CS-92-131, 1992. Ph.D. dissertation.

[5] Olsen, D. Propositional production systems for dialogue description. In *Human Factors in Computing Systems: Proceedings of CHI'90.* ACM Press, pp. 57-63, 1990.

[6] Olsen, D., Monk, A. and Curry, M. Algorithms for automatic dialogue analysis using propositional production systems. Preprint of journal article accepted for publication, 1994.

[7] Clarke, E., Emerson E. and Sistla, A. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions* on *Programming Languages and System*, vol. 8, no. 2, pp. 244-263, Apr. 1986.

[8] Burch, J., Clarke, E., McMillan, K., Dill, D. and Hwang, J. Symbolic model checking: 10^{20} states and beyond. In *Lecture Notes in Computer Science*, Springer-Verlag, 1990.

[9] Atlee, J. M. and Gannon, J. State-Based Model Checking of Event-Driven System Requirements. In *IEEE Transactions* on *Software Engineering*, vol. 19, no. 1, pp. 24-40, Jan. 1993.

[10] Campos, S., Clarke, E., Marrero, W., Minea, M. and Hiraishi, H. *Computing Quantitative Characteristics of Finite-State Real-Time Systems*. Carnegie Mellon University, Computer Science Department, Technical Report CMU-CS-94-147, May 1994.

[11] Harel, D. Statechart: a visual formalism for complex systems. In *Science of Computer Programming*, vol. 8, no. 3, pp. 231-274, Jun. 1987.

[12] Harel, D. On Visual Formalisms. In *Communications of the ACM*, vol. 31, no. 5, pp. 514-530, May 1988.

[13] *Statemate 4.5 Analyzer User Reference Manual*. i-Logix, Inc., Aug. 1992.