Algorithms for Automatic Dialogue Analysis Using Propositional Production Systems

Dan R. Olsen, Jr.
Brigham Young University

Andrew F. Monk and Martin B. Curry
University of York

ABSTRACT

The specification of a graphical user interface (GUI), like any other part of a computer system, is an incremental process whereby an outline of the system is systematically developed, evaluated, and revised until it is reasonably complete. This article describes some algorithms and procedures that can be used to automate the analysis of a specification to facilitate this iterative process.

A propositional production system (PPS) is a notation that can be used by designers to describe the high-level behavior of a GUI. Such a description is executable and relatively easy to learn and use. PPSs are a form of state machine; therefore, much of the theory of state machines can be applied to their analysis. PPSs, however, provide the advantage of semiparallel definitions of state transitions. This is important, as dialogue models of modern GUIs allow a large number of simulta-

Dan R. Olsen, Jr. is a computer scientist who has done extensive research in user interface software architectures and is currently a Professor of Computer Science and Department Chair at Brigham Young University. Andrew F. Monk is a Reader in Psychology at the University of York, where he is involved in a variety of research on human–computer interaction. Martin B. Curry is a human factors engineer with interests in human–system interaction design; he is currently a Senior Research Scientist within the Human–System Interaction Group, Sowerby Research Centre, British Aerospace plc.
CONTENTS

1. THE NEED FOR FORMALISMS AND ABSTRACTION IN HUMAN--COMPUTER INTERACTION

2. DIALOGUE MODELING: EXAMPLE OF HOW A PROPOSITIONAL PRODUCTION SYSTEM SPECIFICATION CAN BE USED
   2.1. Task Model
   2.2. PPS Dialogue Model
   2.3. Evaluation and Analysis

3. MORE ABOUT PPS

4. QUESTIONS THAT CAN BE ASKED OF A PPS DIALOGUE MODEL
   4.1. Verification: Are There Rules in Conflict?
   4.2. Reachability: Can Preconditions for a Rule Be Reached From Some Starting State?

5. PPS ANALYSIS ALGORITHMS
   5.1. Computational Complexity
   5.2. Operations on Condition Vectors
   5.3. Condition Vector Lists
   5.4. Set Definitions of Rule Transitions
   5.5. Detecting Conflict
   5.6. Special Sets of States
   5.7. State Reachability Algorithms
       Special Issues in State Graphs
       Simple State Reachability
       Input Path Analysis
       State Set Reachability
       Dead State Reachability
   5.8. Maximum Minimal Paths to Reach Any Given Rule
   5.9. Rule Set Connectedness

6. CONCLUSIONS

neously available inputs leading to very large state spaces. By dealing in sets of states, a PPS makes the problem of describing the potentially exponential number of state transitions tractable. This article discusses how this innovation can lead to efficient algorithms for analyzing a dialogue model for properties such as task completeness, reversibility of effect, accessibility, connectedness, and avoidance of deadlock.

1. THE NEED FOR FORMALISMS AND ABSTRACTION IN HUMAN--COMPUTER INTERACTION

The design of a graphical user interface (GUI), like the design of any other complex artefact, cannot proceed in one step. The designer has to
try out ideas. Some will be rejected; others will be modified and expanded until a full specification emerges. During this process, models of the eventual GUI are built. These models might take the form of pencil sketches of screens, English-language descriptions of the behavior of the system, or partial prototypes. A model differs from the full implementation in that the model is easy to change and describes only some aspect of the design. Sketches, English-language descriptions, and partial prototypes are all created to be changed and discarded after they have served their purpose, which is to help the designer reason about the design. Models also serve as a medium of communication with other designers and users. Gould, Boies, Levy, Richards, and Schoonard (1987) described a variety of techniques by which the design for the messaging system for the Los Angeles Olympics were, in our terms, modeled. Gould et al. reported that one of the most effective techniques used was writing a short user guide describing how various tasks would be achieved using the system. This was done before any code was written. It served as an effective medium for communicating with users, and several modifications were made. The refined user guide also became an important document by which the designers coordinated their work.

Documentation and prototypes are concrete models of the design. It is also possible to produce abstract models. Command language grammar (CLG; Moran, 1981) encourages the designer to build a series of models of the design at several different levels of abstraction. The most abstract is the task-level description. There are also semantic-level and syntactic-level descriptions. Physical actions and display changes are specified only when one gets to the interaction-level description. The value of an abstraction is that it allows the designer to reason about some specific aspect of the design without the distraction of all the other details. As GUIs get more complex, this becomes more and more important. It is all too easy to get bogged down in the minutiae of surface details and so be distracted from the overall behavior of the system and how it maps onto the user's work objectives. Abstraction facilitates reasoning and communication--the two reasons for building models.

The other advantage of abstraction is that it permits a degree of formality and thus proof and analysis. Choosing an abstraction with a known mathematical basis makes it possible to exploit a large body of mathematical knowledge. For example, should a specification be represented as a state-transition network, it is possible to use the large body of mathematical knowledge from formal languages and graph theory to analyze and prove things about the specification. One might want to prove that it is possible to get from some starting state to a critical end-state. Similarly, one might want to analyze the complete state-transition diagram in order to see if there are any blind alleys or loops from which there is no escape.

There is a growing body of work on the formal specification of user interfaces (see, e.g., Harrison & Thimbleby, 1990). This work has made
use of established notations for formal specification—such as CSP, Z, and algebra. This work is still short of being exploitable in an industrial setting because the notations are difficult and time consuming (see, e.g., Monk, Curry, & Wright, 1994). Although high cost in terms of training and effort is justifiable in a research setting, these techniques have yet to demonstrate a payback that would justify their use in a real design environment.

The cost effectiveness of formal methods has to be seriously considered. Even in the area of safety-critical systems, in which the additional cost of applying these methods can be justified by the need to prove that a specification meets its requirements, there is evidence of significant practical problems (Monk et al., 1994). The best known formal methodology for user interface design, Moran's (1981) CLG, has similarly been criticized for its complexity and the large amount of effort required to translate between the different levels (Bellotti, 1988; Sharratt, 1987).

The approach taken in this article is to minimize the cost of using a formalism. This can be achieved by choosing a notation that should be easy for a designer to learn and to use, by limiting the scope of formal analysis, and by providing computer support for proof and analysis.

**Ease of Use and Learning.** The notation used in this article is the propositional production system (PPS; Olsen, 1990). PPSs have a sound mathematical basis in propositional logic and formal languages but should be easier for the average programmer to understand than a notation such as CSP or Z.

**Scope of Analysis.** The problem that formal techniques such as CLG have set themselves is to support the complete design process. This is very ambitious. Formal modeling can still be valuable when applied to some limited part of the design. The resulting relative simplicity makes the technique several orders of magnitude easier to apply. Of course, the potential payoff must also be reduced, but much less so. This article illustrates this approach as it might be applied to modeling the coarse-grain human–computer dialogue (HCD) for a user interface.

**Computer Support.** Most formal notations require analysis by hand. The main body of this article is concerned with the automatic analysis of PPS specifications. We describe algorithms that will identify potential usability problems in the specification—such as points at which the user interface might deadlock, tasks that are unreasonably difficult to complete, and effects that cannot be undone. Being able to automatically detect such problems early in the design process is a considerable advantage.

The next section has the dual purpose of (a) illustrating a methodology for using a PPS specification and (b) introducing the notation. Section 3
2. DIALOGUE MODELING: EXAMPLE OF HOW A
PROPOSITIONAL PRODUCTION SYSTEM
SPECIFICATION CAN BE USED

Production systems have a long history of use for modeling HCD. Early
user interface management systems (see Green, 1985) envisaged dialogue
specification in terms of event handlers in which the effect of an event is
governed by and expressed in terms of system states. More recent tools for
the generation of user interfaces—such as the UIDE project (see, e.g.,
Foley, Kim, Kovacevic, & Murray, 1991) and the GARNET project (see,
e.g., Myers et al., 1990)—similarly use rules with preconditions and
postconditions to describe the behavior of a user interface. These tools and
methods have as their objective the specification of the low-level HCD,
as the tools are to automatically generate a user interface from this
specification.

A strong case can be made for specifying HCD at a much higher level
of abstraction. The argument is that the commercial software designer's
real problem is not to design the low-level, fine-grain HCD, as this is
generally already specified by the style guide or tool box selected for the
project. The difficult problem is to design the high-level, coarse-grain
HCD in such a way that it matches the way users approach their work
objectives. This article proposes a method by which HCD can be speci-
fied, using a PPS, at the same level of abstraction as the task model.
Because they are at the same level of abstraction, the dialogue model and
the task model can be validated one against the other. After this has been
accomplished, the abstract dialogue model can then be transformed into a
full specification by applying a style guide to elaborate the fine-grain
details.

The examples to be developed here illustrate this approach, although
the algorithms described in the body of the article for the automatic
analysis of PPS specifications could be applied to a high- or low-level
specification of a user interface.

2.1. Task Model

The following example is drawn from a case study based on a realistic
design task—specifying a GUI that integrates the functions of the two
text-based user interfaces for recording and controlling the flow of prod-
ucts through a food distribution warehouse. This case study was developed
Observation of and interviews with the operators in the warehouse lead to
Figure 1. Work objective decomposition (WOD) for work of an operator in the food distribution warehouse.

1. Delivery logged
   1.1. Tally cards generated
      1.1.1. Delivery note obtained from driver
      1.1.2. Gate house record number entered
      1.1.3. Supplier details entered
      1.1.4. Products entered with quantities
      1.1.5. Printing of tally cards requested
   1.2. Delivery note validated
      1.2.1. Quantities agreed with warehouse person
      1.2.2. Printing of confirmation of delivery requested
      1.2.3. Confirmation of delivery given to driver
   1.3. Locations of delivery logged
      1.3.1. Locations obtained from warehouse person
      1.3.2. Locations entered
   1.4. Delivery record sent to central database
2. Goods allocated to stores
3. Picking notes generated
4. Store delivery note printed
5. Grid checking note printed

Figure 2. User exceptions for WOD shown in Figure 1.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Exception Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>Typing error noticed in delivery record (at 1.1, 1.2, or 1.3)</td>
</tr>
<tr>
<td>Problem 2</td>
<td>Delivery amounts do not tally (at 1.2)</td>
</tr>
<tr>
<td>Interruption 1</td>
<td>Priority delivery must be processed immediately (at 1.1, 1.2, or 1.3)</td>
</tr>
</tbody>
</table>

the top-level specification of their work objectives, shown in Figure 1. This is essentially an “ecological” subgoal hierarchy, as each objective is given as the state of the world to be achieved (e.g., Objective 1 is to be read as “to have this delivery logged,” and Objective 1.1 is to be read as “to have the tally cards generated”).

The objectives are numbered for convenience, but the numbers are not meant to imply that the objectives have to be carried out in any particular order. In a conventional hierarchical task analysis, this information would be appended as a “plan.” We choose not do this, as this information is implicit in the dialogue model part of the specification.

Along with specifying a work objective decomposition (WOD), the task model specifies user exceptions and typical scenarios of use. Typical scenarios of use are narrative accounts of typical sequences of tasks, including specific examples of the data involved (e.g., real delivery notes, tally cards). User exceptions are points in the dialogue at which the ideal sequence of events as represented in the WOD might be disrupted because the user is interrupted or makes a mistake. The important user exceptions for the warehouse operators are specified in Figure 2. The
numbers in parentheses refer to the WOD in Figure 1 and give the scope of each user exception.

2.2. PPS Dialogue Model

Given the description of the user's work provided by the task model, a designer might start to specify a dialogue model for the new GUI that is to support this work. Let us assume that the whole delivery record, including location of goods, is recorded on a single electronic form accessed by some user event. On the basis of the WOD, one can then identify some high-level user events:

- CreateNewDeliveryRecord
- SelectDeliveryRecord_1
- EditDeliveryRecord_1
- ConfirmAsCompleteDeliveryRecord_1

- SelectDeliveryRecord_1 and EditDeliveryRecord_1 cover Objectives 1.1.3, 1.1.4, and 1.3.2. The prefix indicates a user-generated event. These are the focus of each PPS rule. These rules attach preconditions and postconditions to each user event. The preconditions specify when the user event will have some specified side effect. For example, the key user event is editing a delivery record. Only one record can be edited at a time, and so a precondition is that that record is currently selected. Preconditions are listed on the left-hand side of the rule before the user event and its associated side effect as follows:

    #DeliveryRecord_1selected
    -EditDeliveryRecord_1 --> (record and display changes)

Postconditions record changes to the state of the dialogue model that is the preconditions for the next rule to fire. These are recorded on the right-hand side of the rule, after the side effect, as shown here:

    No preconditions
    -SelectDeliveryRecord_2 --> (record and display changes)
    #DeliveryRecord_2selected

The conditions in a PPS, signified here by the prefix #, are put into "fields." The conditions in a field are mutually exclusive, so setting one unsets the condition currently set. By putting #DeliveryRecord_1selected and #DeliveryRecord_2selected in the same field, the designer ensures that setting one as the postcondition of a rule firing will always unset the other.

The following field and six rules model the situation in which three delivery records are in progress:
Field
F1. SelectedRecord(#DeliveryRecord_1selected,
    #DeliveryRecord_2selected, #Delivery
    Record_3selected)

Starting state
#DeliveryRecord_1selected

Rules
R1. No preconditions
   ↓-SelectDeliveryRecord_1  -->  (record and display changes)
      #DeliveryRecord_1selected

R2. #DeliveryRecord_1selected
   ↓-EditDeliveryRecord_1  -->  (record and display changes)
      No postconditions

R3. No preconditions
   ↓-SelectDeliveryRecord_2  -->  (record and display changes)
      #DeliveryRecord_2selected

R4. #DeliveryRecord_2selected
   ↓-EditDeliveryRecord_2  -->  (record and display changes)
      No postconditions

R5. No preconditions
   ↓-SelectDeliveryRecord_3  -->  (record and display changes)
      #DeliveryRecord_3selected

R6. #DeliveryRecord_3selected
   ↓-EditDeliveryRecord_3  -->  (record and display changes)
      No postconditions

R1, R3, and R5 are equivalent and allow the user to select each of the three delivery records. R2, R4, and R6 similarly allow the user to edit each of the three records. It would be possible to write a general propositional form that matches any delivery record, but, for simplicity, we use this lengthier but straightforward notation.

The possible interactions a user might have can be traced through this model as follows:

1. R1, R3, and R5 have no preconditions. This supports Interruption 1 (Figure 2), as it allows the user to select a record at any time. The starting state is set arbitrarily as having Delivery Record 1 selected, and so R2 also has its precondition satisfied. So, at this point, the user has available the actions:

   ↓-SelectDeliveryRecord_1
   ↓-EditDeliveryRecord_1
   ↓-SelectDeliveryRecord_2
   ↓-SelectDeliveryRecord_3

2. Selecting the event ↓-SelectDeliveryRecord_2 from this list will have the effect of activating R3, which has the postcondition
#DeliveryRecord_2 selected. Setting this condition has the effect of unsetting #DeliveryRecord_1 selected, as they are in the same field. At this point in the dialogue, R1, R3, R4, and R5 have their preconditions satisfied, and so the user has available the actions:

→ SelectDeliveryRecord_1
→ SelectDeliveryRecord_2
→ EditDeliveryRecord_2
→ SelectDeliveryRecord_3

3. Selecting → EditDeliveryRecord_2 from this list has no effect on the state of the dialogue model. Of course, the user event has an effect as recorded in the side effect (record and display changes), but the list of user events available is unchanged.

Figure 3 presents the next stage in the elaboration of this design. R7, R8, and R9 are equivalent and model the effect of confirming a delivery as complete. This has the side effect of updating the central database used to allocate goods to stores. After this has been done, we do not wish to allow the user to edit the record, as this would lead to discrepancies between this central record and the local record held in the warehouse. These rules have a postcondition (e.g., #DeliveryRecord_1 confirmed) that unsets the starting state #NotDeliveryRecord_1 confirmed because they are in the same field. #NotDeliveryRecord_1 confirmed has also been added as a precondition to R2, so that user event → EditDeliveryRecord_1 is no longer available after the delivery has been confirmed.

A PPS specification of this kind is relatively easy to read and to reason about. Also, it makes explicit the design decisions taken. For example, R10 has the postcondition #NewDeliveryRecord selected recording a decision that a new record should be selected immediately after it is created, thus deselecting the currently selected record. Figure 3 is still only a partial specification. The designer would go on to add rules for each of the other components of the task model.

2.3. Evaluation and Analysis

At each stage in this incremental process, the designer can evaluate the dialogue model against the task model. This can be greatly facilitated if the dialogue model is animated. This requires a software tool (see, e.g., Monk & Curry, 1994) that is able to record what conditions are set and use this record to display a list of user events that are available because the preconditions of a rule are satisfied. The designer can then examine the effect of one of these user events by selecting it. This changes the conditions set according to the postconditions of the rule, and a new list of available user events is displayed. The designer
**Figure 3.** Partial dialogue model for food distribution warehouse example.

**Fields**

F1. SelectedRecord (#NewDeliveryRecordselected, #DeliveryRecord_1selected, #DeliveryRecord_2selected, #DeliveryRecord_3selected)

F2. ConfirmedDR1 (#DeliveryRecord_1confirmed, #NotDeliveryRecord_1confirmed)

F3. ConfirmedDR2 (#DeliveryRecord_2confirmed, #NotDeliveryRecord_2confirmed)

F4. ConfirmedDR3 (#DeliveryRecord_3confirmed, #NotDeliveryRecord_3confirmed)

**Starting state**

#DeliveryRecord_1selected, #NotDeliveryRecord_1confirmed, 
#NotDeliveryRecord_2confirmed, #NotDeliveryRecord_3confirmed

**Rules**

R1.

No preconditions

↓ SelectDeliveryRecord_1 --> (record and display changes)

#DeliveryRecord_1selected

R2.

#DeliveryRecord_1selected, #NotDeliveryRecord_1confirmed

↓ EditDeliveryRecord_1 --> (record and display changes)

No postconditions

R3.

No preconditions

↓ SelectDeliveryRecord_2 --> (record and display changes)

#DeliveryRecord_2selected

R4.

#DeliveryRecord_2selected, #NotDeliveryRecord_2confirmed

↓ EditDeliveryRecord_2 --> (record and display changes)

No postconditions

R5.

No preconditions

↓ SelectDeliveryRecord_3 --> (record and display changes)

#DeliveryRecord_3selected

R6.

#DeliveryRecord_3selected, #NotDeliveryRecord_3confirmed

↓ EditDeliveryRecord_3 --> (record and display changes)

No postconditions

R7.

#DeliveryRecord_1selected, #NotDeliveryRecord_1confirmed

↓ ConfirmAsCompleteDeliveryRecord_1 --> (update central database)

#DeliveryRecord_1confirmed

R8.

#DeliveryRecord_2selected, #NotDeliveryRecord_2confirmed

↓ ConfirmAsCompleteDeliveryRecord_2 --> (update central database)

#DeliveryRecord_2confirmed

R9.

#DeliveryRecord_3selected, #NotDeliveryRecord_3confirmed

↓ ConfirmAsCompleteDeliveryRecord_3 --> (update central database)

#DeliveryRecord_3confirmed

R10.

No preconditions

↓ CreateNewDeliveryRecord --> (create and display new record)

#NewDeliveryRecordselected
can run through the scenarios in the task model and check that the model allows the user to generate the right user events at each point in the dialogue. In this way, the designer can identify points in the dialogue at which a task cannot be completed because user events that should be available are not. One can also identify points in the dialogue at which user events are available when they should not be, as in the case of editing confirmed delivery records in the example presented earlier.

A more exhaustive search of the design space is required to evaluate the WOD. Finally, the designer needs to go through the WOD with the user exceptions to check that the dialogue model allows the appropriate actions to be taken. Analysis of a dialogue model through observation of its behavior in this way is useful for gaining an understanding of the design. However, a designer might ask several questions that are difficult or tedious to answer by executing the specification. This is where the automatic analyses specified in Sections 4 and 5 come in. Before going to these sections, we must give some further details of PPS.

3. MORE ABOUT PPS

Olsen (1990) introduced PPS. PPS is similar to a state-transition diagram in that it consists of rules specifying possible state transitions. State-transition diagrams are representations with which most programmers are familiar. State-transition diagrams have also been used to model user interfaces. Thimbleby (1993), for example, described how the graph theory routines in the mathematical package Mathematica (Wolfram, 1988) can be used to analyze the user interface to a video recorder. The problem with using any simple state machine to model a modern GUI is that the large number of parallel options leads to a state explosion that makes specification and analysis impractical. PPS solves this problem by working at the granularity of state sets, and most of the algorithms presented here avoid the necessity of enumerating the entire state space.

A PPS differs from the traditional state machine in that, rather than having a single state, the state space is broken into fields that are independent of one another. Each field has a set of two or more conditions in which the field can be. Thus, in the specification presented earlier, the field SelectedRecord could be in one of four mutually exclusive conditions (DeliveryRecord contracted to DR):

```
#NewDRselected
#DR_1selected
#DR_2selected
#DR_3selected
```

although the field ConfirmedDR1 could be in one of two mutually exclusive conditions:
Within a field, the conditions are mutually exclusive. A rule that sets \#DR_1confirmed will thus automatically unset \#NotDR_1confirmed and vice versa. Therefore, we can think of a PPS having several independent states (fields) with rules that process these states either in conjunction or independently.

A key to the expressiveness of a PPS is the use of condition vectors to simply describe a wide variety of possible states. The definitions of condition vectors are presented by example in this discussion. A formal treatment of these definitions was given in Olsen (1992). A condition vector is a list of conditions no two of which are from the same field. The representation of multiple states comes from the fields for which no condition is specified. For example, the condition vector

\[(\#DR_2selected, \#DR_1confirmed, \#NotDR_2confirmed, \#NotDR_3confirmed)\]

specifies exactly one state, given the field definitions in the previous section. The condition vector

\[(\#DR_1confirmed, \#NotDR_2confirmed)\]

represents the following set of states:

\[(\#NewDR_selected, \#DR_1confirmed, \#NotDR_2confirmed, \#DR_3confirmed)\]
\[(\#NewDR_selected, \#DR_1confirmed, \#NotDR_2confirmed, \#NotDR_3confirmed)\]
\[(\#DR_1selected, \#DR_1confirmed, \#NotDR_2confirmed, \#DR_3confirmed)\]
\[(\#DR_1selected, \#DR_1confirmed, \#NotDR_2confirmed, \#NotDR_3confirmed)\]
\[(\#DR_2selected, \#DR_1confirmed, \#NotDR_2confirmed, \#DR_3confirmed)\]
\[(\#DR_2selected, \#DR_1confirmed, \#NotDR_2confirmed, \#NotDR_3confirmed)\]
\[(\#DR_3selected, \#DR_1confirmed, \#NotDR_2confirmed, \#DR_3confirmed)\]
\[(\#DR_3selected, \#DR_1confirmed, \#NotDR_2confirmed, \#NotDR_3confirmed)\]

The field definitions define the state space of a PPS. The rules define the set of acceptable transitions in that space. A PPS rule consists of a left-hand condition vector and a right-hand condition vector. For example, R7, presented earlier,

\[\text{#DR_1selected, \#NotDR_1confirmed} \quad \downarrow \text{ConfirmAsCompleteDR_1} \rightarrow \text{update central database} \quad \text{#DR_1confirmed}\]

can be thought of in terms of the two condition vectors:

1. The precondition (\#DR_1selected, \#NotDR_1confirmed).
2. The postconditions (\#DR_1confirmed).
In English, one can describe the operation of this rule thus: If any state in
the set specified by the precondition vector (#DR_1selected,
#NotDR_1confirmed) is active, then the user event ↓-ConfirmAsCompleteDR_1 will lead to a new state in which #DR_1confirmed is set. The latter can be thought of as overlaying the postcondition state vector to produce the new state (see Section 5.1 for definition).

As with any production system, there must be an algorithm specified for
dealing with conflict resolution. This is the case in which more than one
rule can fire after receiving a given user input. In our example, each user
event enters into only one rule, and so this could never happen. It is
possible to imagine situations in which the same user event gives rise to
different effects depending on the state of the dialogue model. In such a
case, there would be one rule for each event–effect mapping. There is then
the further possibility that there are states in which the same user event
could fire two rules.

If two different rules can fire, and they assert mutually exclusive condi-
tions, then there is a problem that must be resolved in some way. When a
user enters an event, it is not acceptable for the dialogue management to
guess what the user wanted done; what the user wanted done must be
clearly determined. Choices are for users to make. Having expressed those
choices by means of some input, the dialogue algorithms must resolve the
contradiction in a deterministic way. In most PPS specifications, there are
no conflicts. The algorithms must, however, have some resolution strat-
 egy, because conflicts are possible.

In a normal PPS, rules are arranged in priority order. Given a particular
state of the PPS, each rule is examined in this order. If the precondition
vector of a rule contains the current PPS state, and the right-hand side of
the rule does not conflict with any rules of a higher priority, then the rule’s
right-hand side is overlaid onto the new state vector. This model of rule
firing allows multiple rules to fire for a given state of the PPS. An alternative
arrangement is to interpret conflict as evidence of a poorly specified
design and to ask the designer to resolve it by making the preconditions
more specific. Either solution is amenable to support by automatic analysis
(see Section 5.3).

The notation used here differs in presentation from that in Olsen’s
(1990) original definition of PPS in treatment of user events and side
effects. Olsen allowed for the possibility of fields of independent user
events that could arise in parallel. For the application of PPS to abstract
dialogue modeling, it is reasonable to assume that all user events are
mutually exclusive. That is, one has to be dealt with before the next can be
considered. This means that it is not necessary to differentiate “input
fields” in this way. Input fields are different from “state fields” (equivalent
to all the “fields” used in this presentation) in that they can occur only on
the left-hand side of a rule. For simplicity, we have separated them from
the preconditions and given them a different prefix. In some of the
analyses to be presented here, the condition vector is taken as including a
user event. In these cases, the user event can be thought of as being a
condition in a field containing all the user events.

The side effects of a rule firing specified here within braces \( \{ \); e.g.,
\{update central database\} \) were described in Olsen’s (1990) orig-
inal paper as “action fields.” Again, Olsen allowed for the possibility of
independent sets of side effects expressed conveniently as fields. Condi-
tions from action fields can occur only on the right-hand side of a rule.
Action conditions do not enter into the analyses considered in this article
and hence are simply treated as side effects of a rule firing.

PPS is similar to event response language (ERL; Hill, 1987), except that
Hill’s (1987) conditions are binary flags (two-condition fields in our terms).
This makes ERL specification a subset of those expressible in a PPS. None
of the ERL work has addressed the issues of automatically analyzing
dialogue to aid designers in creating good specifications.

UIDE (Foley et al., 1991) allows preconditions that are not reasonably
expressed in propositional logic. For example, one of the preconditions for
a rule might be that some variable is greater than some value. This
sophistication increases the expressive power of the notation. However,
the additional complexity introduced makes them less suitable as tools for
specification, as they are harder to reason about. The conflict resolution
algorithm in ERL might make for more concise code when it comes to
building a user interface, but this also means that small changes can have
large, unexpected effects on the behavior of a specification. The use of
nonpropositional preconditions inUIDE is similarly convenient for pro-
gramming but precludes the use of propositional logic and most of the
analyses described here. The algorithms in this article take advantage of
the fact that a PPS is simply a more expressive formulation of state
machines. Because of this, much of the theory of state machines can be
applied, whereas more sophisticated models of computation have been
shown to be intractable for these types of analysis.

The advantages of PPS can be summarized as follows:

1. PPSs provide a concise notation. These semiparallel definitions of
state transitions are much more flexible to use than simple state machines
and provide some distinct advantages in the number of rules required. In
the case of the very simple example presented earlier, there are \( 4 \times 2 \times 2 \times 2 = 32 \) possible states. Our specification describes the transitions be-
tween these possible states with 10 rules. In general, the number of PPS
rules needed to describe a user interface will be linear in the number of
simultaneously available inputs, whereas the number of transitions will be
exponential with a normal state machine.

2. A PPS is expressed in terms of state, events, and if–then rules. The
majority of people working within a software development environment
are familiar with these concepts.
3. A PPS can be animated. That is, given the appropriate software tool, it is possible to execute productions and observe the behavior of the specification.

4. A PPS is amenable to automated analyses. The questions that one could ask of these analyses are spelled out in the next section.

4. QUESTIONS THAT CAN BE ASKED OF A PPS DIALOGUE MODEL

Section 2 illustrates how a PPS could be used to construct a dialogue model and to validate it against a model of the user's task. The process is iterative in that the dialogue model is evaluated against the task model by animating it using a software tool. This article is concerned with various additional facilities that could be provided in such a software tool to aid the design process by analyzing the dialogue design in various ways. In all prior work with this method (see, e.g., Monk & Curry, 1994), analysis of the dialogue is done by the designer stepping through an execution of the dialogue. The manual exploration of the design is helpful but can become tedious in large designs. These manual explorations are the inspiration for the automatic analysis presented in this work.

A major advantage of having a formal model of the user dialogue is that the model can be analyzed automatically to answer various design questions that are difficult to answer reliably in handwritten notations or with less formal, paper-only dialogue models. Before proceeding to the analysis algorithms, it is useful to review the two basic kinds of questions for which we can provide automatic answers. These are questions of verification and reachability.

4.1. Verification: Are There Rules in Conflict?

Several checks need to be made to verify a PPS dialogue model. There should be no duplicate conditions (i.e., no conditions with the same name in different fields). There should be no conditions specified in the rules that are not defined in the fields. There should be no rules with two preconditions from the same field, as such a rule could never fire. Similarly, there should be no rules with two postconditions from the same field.

These checks are fairly basic. If any are violated, the PPS remains undefined. A more subtle verification question concerns whether any dialogue rules are in conflict. There are two issues here. The first is a definition of what it means for two rules to be in conflict, and the second is a model for how to handle the conflicts.

There are two definitions of rule conflict that have been proposed for a PPS. The original PPS work defined a weak conflict, which said that two rules are in conflict if there exists a state for which each rule would assert
some conflicting condition (i.e., one rule asserts a condition that is in the same field as a different condition from the same field asserted by the other rule). For the PPS algorithm to function at all, the weak conflicts must be resolved. The original PPS resolved weak conflict by prioritizing the rules. Whenever two rules are in conflict, the lower priority rule is blocked and is not allowed to fire.

An alternative resolution is to not allow weak conflict and instead require the designer to resolve the problem. It is assumed that each rule was created by a designer for a specific purpose. If two rules are in weak conflict, then the purpose of one of the rules cannot be completely fulfilled because it will be overridden by the other rule. Our current thinking is that it is important to point out these situations so that the designer can explicitly account for them rather than simply have some default action occur.

Resolution of weak conflict is essential for the proper functioning of the PPS execution algorithm. A strong-conflict model says that two rules are in conflict if there exists a state in which both rules can fire. Strong-conflict checking is motivated by considerations of good design specifications. If one assumes that, in a good PPS design, each rule can be understood independently, then strong conflict is required. In the presence of strong-conflict checking, one can look at a single rule alone and know all that can possibly happen in the situation described by that rule. If only weak-conflict checking has been done, there might be other rules that can fire that do other things. The weak-conflict model allows for more compact rule sets in many circumstances. We present algorithms for both models here. A possible implementation of these in a design tool would be to forbid weak conflict while providing highlights of the existence of strong conflict.

4.2. Reachability: Can the Preconditions for a Rule Be Reached From Some Starting State?

The state of the dialogue is characterized by the conditions of each of the fields. An obvious question to be asked of any state machine is whether one can reach one state, the target state, from another, the starting state. As has been pointed out, modern GUIs make available a large number of different user inputs at any point in the dialogue, and this gives rise to very large state spaces and exponential numbers of possible transitions between these states. Traditional state machine analysis would enumerate all the states and their transitions. This would create an excessive computational burden for a design tool. A PPS solves this problem by dealing in sets of states. Within the state space, the important target states are the state sets specified by the preconditions of each rule that define when a particular user event is available. This is because tasks are defined in terms of user events and their associated side effects; thus, a designer will want to know
whether it is possible to get from some starting state to a state at which a rule with the desired side effect could fire.

An example makes this clear. In Figure 3, R7 has the side effect of updating the central database. As such, it marks an important step in the completion of the tasks specified in Figure 1. For this reason, the designer will want to know that it is possible to reach a state at which the user event for the rule (↓-ConfirmAsCompleteDeliveryRecord_1) is available. The set of states for which this is true is of course given by the preconditions for the rule—in this case, #DeliveryRecord_1selected, #NotDeliveryRecord_1confirmed. The question could be answered by animating the dialogue model. Some start state would be set, and then various sequences of user events would be tried out to check that there is indeed a way of getting the dialogue model from the start state to a state at which these preconditions are satisfied. Although this kind of manual exploration is sometimes necessary to gain an understanding of the dialogue model, there will be several reachability questions more effectively asked by a designer automatically.

Knowing that one can reach one state from another verifies a design but does not necessarily show it is a good one. A designer might like to know how much user effort is required to do this. One measure of user effort is the minimal number of user events required to make the transition. Coarse-grain user events such as ↓-CreateNewDeliveryRecord and ↓-SelectDeliveryRecord_1 might differ very much in the effort required of the user. Another possibility, then, is to assign costs to user events, or, as Bleser and Foley (1982) proposed, costs can be assigned to transitions between user events. For example, the cost of a mouse-based event followed by a keystroke-based event would be higher than the cost of a keystroke followed by a keystroke because less hand movement is required.

Figure 4 sets out several reachability questions that could be asked, defined in terms of some starting state or set of states and some target rule or set of rules.

Task Completeness. This question asks whether there is a path through the dialogue model from the state at initialization to a state at which some named rule can fire because its preconditions are satisfied. As discussed earlier, one can ask whether a task can be completed or what is the minimum “cost” of completing the task.

Task Connectedness. Many tasks require several rules to be fired before they are completed. In such cases, one might need to take these rules in pairs and consider whether or how reachable one is from the other. Here we are dealing with a set of start states—that is, the states
that could obtain after the first rule has fired. The question is how easy it is to get from all these states to a state at which the second rule has its preconditions satisfied. For example, one might want to set up an automatic analysis to check that R2 in Figure 3 (↓-EditDeliveryRecord_1) is always available immediately after R1 (↓-SelectDeliveryRecord_1) has fired.

Reversibility of an Effect. The undo function (and the general ability to undo the effect of some action) is one of the most important features of modern GUIs. This is because it allows the user to explore the use of the system without fear of doing something irreversible. A designer will want to examine each rule, in turn, for reversibility of effect—that is, (a) whether there is a rule the side effect of which reverses the side effect and (b) how easy it is to reach a state at which that rule is available. It might not always be computationally possible to provide easy reversibility, but the designer should at least be aware of where such problems exist. Such an analysis of the rules in Figure 3 would show that the effects of R1 to R6 are easily reversible. A delivery record can be reselected or reedited in the next user event. R7 to R9 are not reversible, and this is as it should be. The effect of R10 (↓-CreateNewDeliveryRecord) is not reversible, as there is no way to delete a delivery record, and this indicates a weakness in the current design.

Reversibility of Mode. As the state determines what user events are available, it can be thought of as determining mode. A user might want to
reverse the effect of a rule in terms of getting back to the same choice of
user events.

*Accessibility of a Rule.* A designer might want to nominate rules that
should be easily accessible from any point in the dialogue. These will be
rules with important side effects, such as displaying help or saving a
document. This question involves finding the worst case, in terms of the
largest minimal cost for making such a rule available.

*Rule Set Connectedness.* On the assumption that all rules are inserted
for a purpose, the designer will want to know that there are no rules that
could never fire.

*Deadlock.* The designer will want to know that it is not possible to
reach a state from which there is no escape because it satisfies the precon-
ditions of none of the rules. Such a situation would, of course, be very
annoying to a user. Because of the complexity of the state space of a
dialogue, it might be hard for designers to recognize when this might
occur. This problem is particularly acute when designers are focused on
how the dialogue is supposed to be used, rather than on the ways in which
it might be incorrectly used.

As can be seen from Figure 4, many of these questions are functionally
similar. For example, reversibility of effect is a special case of task connect-
edness, and rule set connectedness is a generalization of task complete-
ness. This will be used in the algorithms for automatically answering these
questions, to be developed here.

5. **PPS ANALYSIS ALGORITHMS**

The first three subsections in this section can be regarded as preliminari-
ies. They define basic operations that are used in the remaining five
subsections, which describe algorithms for analyzing a PPS specification.

5.1. **Computational Complexity**

As mentioned earlier, the primary benefit of the PPS model is that a
combinatorically large number of states and transitions can be represented
by a few fields, conditions, and rules. This is the primary source of the
expressive power of a PPS. The combinatoric size of the domain, however,
raises serious problems when trying to automatically analyze a specifica-
tion. We would like the complexity of the analysis to be roughly on the
same order as the number of fields, conditions, and rules rather than
having a complexity related to the actual state space.
Take, for example, the simple problem of $N$ inputs, each of which should be entered exactly once but in any order. The PPS formulation requires $N$ fields, $2N$ conditions, and $N + 1$ rules. The corresponding state machine has $2^N$ states with an even larger number of transitions. We would like our analysis of this PPS to take on the order of $N$ steps to reach its conclusions, rather than $2^N$ steps, as would be required by the corresponding state machine.

Path algebras (Alty, 1984) were proposed as a mechanism for analyzing state machines. All the proposed algorithms assume that the state space has been completely enumerated. In the PPS model, this would increase the computational complexity of the problem well beyond acceptable limits for all but very small designs. Through the discussion of these algorithms, the issue of not exploring or completely enumerating all possible states will be a recurring theme.

5.2. Operations on Condition Vectors

In Section 3, the concept of a condition vector was defined as a list of conditions no two of which are from the same field. A condition vector defines a set of states that is the set of states in the complete state space that include those conditions. Let us say there are four fields:

\[ F1(A, B, C) \]
\[ F2(M, N) \]
\[ F3(T, U) \]
\[ F4(X, Y) \]

The complete state space is:

\[ (A, N, T, X), (A, N, T, Y), (A, N, U, X), (A, N, U, Y), \]
\[ (B, M, T, X), (B, M, T, Y), (B, M, U, X), (B, M, U, Y), \]
\[ (B, N, T, X), (B, N, T, Y), (B, N, U, X), (B, N, U, Y), \]
\[ (C, M, T, X), (C, M, T, Y), (C, M, U, X), (C, M, U, Y), \]
\[ (C, N, T, X), (C, N, T, Y), (C, N, U, X), (C, N, U, Y) \]


**Intersection.** We can intersect the sets of states represented by two condition vectors by merging the lists of conditions from both vectors. For example, if we intersect $(A, M)$ with the condition vector $(M, T)$, the resulting state set is $(A, M, T, X)$, $(A, M, T, Y)$, which is described by the condition vector $(A, M, T)$. 
If two condition vectors contain conflicting conditions from the same field, then the intersection is empty because no state can satisfy both conditions. In later discussions, we use the predicate \textit{intersects}, which is true if the intersection of two condition vectors is nonempty.

\textbf{Complement.} A second operation that is needed in the following discussion is the complement of a condition vector. A condition vector such as \((A, M)\) is equivalent to the logical statement \(A\) and \(M\). The complement \(\sim (A\ \text{and}\ M)\) is derived using DeMorgan's law to be \((\sim A\ \text{or}\ \sim M)\). The complement of a given condition is the "or" of all the other conditions in its field. Because condition vectors represent conjunctions only, the complement of a condition vector is a list of alternative condition vectors. For example, \(\sim (A, M)\) is \((B)\ (C)\ (N)\).

When there are several condition vectors in a list, as was just the case, they represent the union of their state sets—for example:

\[
\begin{align*}
(B) &= (B, M, T, X), (B, M, T, Y), (B, M, U, X), (B, M, U, Y), \\
     & \quad (B, N, T, X), (B, N, T, Y), (B, N, U, X), (B, N, U, Y) \\
(C) &= (C, M, T, X), (C, M, T, Y), (C, M, U, X), (C, M, U, Y), \\
     & \quad (C, N, T, X), (C, N, T, Y), (C, N, U, X), (C, N, U, Y) \\
(N) &= (A, N, T, X), (A, N, T, Y), (A, N, U, X), (A, N, U, Y), \\
     & \quad (B, N, T, X), (B, N, T, Y), (B, N, U, X), (B, N, U, Y), \\
     & \quad (C, N, T, X), (C, N, T, Y), (C, N, U, X), (C, N, U, Y)
\end{align*}
\]

so

\[
\sim (A, M) = (B)\ (C)\ (N) = \\
(\sim A, N, T, X), (\sim A, N, T, Y), (\sim A, N, U, X), (\sim A, N, U, Y), \\
(\sim B, M, T, X), (\sim B, M, T, Y), (\sim B, M, U, X), (\sim B, M, U, Y), \\
(\sim B, N, T, X), (\sim B, N, T, Y), (\sim B, N, U, X), (\sim B, N, U, Y), \\
\]

As can be seen in this example, one of the problems with complementing a condition vector is that it can cause a potentially explosive number of condition vectors in the result.

\textbf{Difference.} Using the intersect and complement operators, one can define a difference operator that computes a set of condition vectors that describe the set difference of the two condition vector sets. This is defined as \(V1 - V2 = V1 \text{ intersect} \sim V2\).

We define difference because it has less of a tendency to explode than complement. This is due to the fact that many of the condition vectors in \(\sim V2\) have no intersection with \(V1\) and therefore can be discarded.

\textbf{Overlay.} This operator on condition vectors overlays one condition vector with another. This operation is basic to the firing of rules. The operation \(V1 \text{ overlay} V2\) will create a new condition vector that has all the conditions of both \(V1\) and \(V2\). When conditions in \(V1\) are part of the
same fields as conditions in \( V_2 \), the conditions in \( V_1 \) take precedence—
that is, suppose that we have two condition vectors \( V_1 = (A, M) \) and \( V_2 = (N, T) \): \( V_1 \) overlay \( V_2 = (A, M, T) \).

5.3. Condition Vector Lists

Most of the algorithms described in this article use only condition vectors rather than the more expensive condition vector lists. The set defined by a condition vector list is simply a union of the sets defined by each condition vector. Based on this, we can derive operations on condition vector lists directly from those defined on condition vectors. The extensions of the operations from condition vectors to condition vector lists is not discussed in this article because the extensions are easily derived and would complicate the presentation.

A problem with condition vector lists lies in their simplification. For example, if a condition vector list contains the vectors \((A, M)\) and \((A, M, T)\), only the vector \((A, M)\) is required to represent the set, because \((A, M, T)\) represents a subset of the states described by \((A, M)\). The simplification problem is actually more complicated than this example. There is a variety of algorithms from logic and machine learning that were proposed for this purpose (Martinez & Campbell, 1991). True minimization of the list is not required. All that is needed is to simplify the vector list to prevent rampant redundancy, which would render our algorithms useless. The reader is referred to these works for algorithms that reduce redundancy in such lists.

5.4. Set Definitions of Rule Transitions

The original definition of rule application was defined on individual states. A rule was selected and applied to a given state. This needs to be expanded for our analysis algorithms so that rules can be applied to state sets.

A rule \( R \) can be defined as having the following components:

\[
R.lhs \\
R.\downarrow{-ue} \rightarrow R.(se) \\
R.rhs
\]

\( R.lhs \) is the state set representing the preconditions for the rule. \( R.\downarrow{-ue} \) is the user event, and \( R.(se) \) is the side effect of the rule firing. \( R.rhs \) is the vector list representing the postconditions.

Firing the rule causes a state transition. This state set transition will be used extensively in the state reachability algorithms. There are two forms that are required. The first is a forward transition, which models normal rule application. The second is a reverse transition, which works through the rules backward.
**Forward Transitions.** The forward transition takes a state set description and returns the set of states that can be reached by applying exactly one rule. Note that, in doing this, we do not consider the user events (input conditions) or the side effects (semantic conditions). We can first define a forward transition given a single condition vector \( V \) and a given rule \( R \):

\[
\text{ForwardTransition}(V, R) \\
\quad \{ \\
\quad \text{Antecedent} = R.lhs; \\
\quad \text{Consequent} = R.rhs; \\
\quad \text{if} \ V \ \text{intersects} \ \text{Antecedent} \ \text{then} \\
\quad \quad \text{Result} = \text{Overlay}( \ \text{Consequent}, \ \text{Intersection}(V, \ \text{Antecedent})) \\
\quad \text{else} \\
\quad \quad \text{There is no result} \\
\quad \} 
\]

For a given condition vector, we need to apply all the rules to that vector and then union the results together. This yields the set of states that can be reached directly from the set defined by the condition vector. If the input is a condition vector list, we perform this operation for all vectors in the list and union all the results together. This is where simplification of condition vector lists becomes important, because overlapping and redundant condition vectors might be added to the list.

**Reverse Transitions.** Given a set of states, computing the reverse transition yields the set of states that could lead to these states by applying one rule. Reverse transitions sets are computed by reversing the right- and left-hand sides of a rule. This is not quite sufficient, however. What is desired is to determine all states that could produce a state in \( V \) by the application of rule \( R \). Given the fields presented earlier, the rule

\[
(A, M) \\
\downarrow \text{ue} \rightarrow R\{se\} \\
(N, T)
\]

can be used to illustrate a reverse transition.

In computing a reverse transition from some condition vector \( V \), we need to restrict ourselves to those states that can be produced by the rule. This is done by intersecting \( V \) with the consequent (right-hand side) of the rule. We also need to take into account the antecedent (left-hand side) of the rule. The simplest approach would be to overlay the antecedent. This would yield:

\[
\text{Antecedent} = R.lhs; \\
\text{Consequent} = R.rhs; \\
\text{if} \ V \ \text{intersects} \ \text{Consequent} \ \text{then} \\
\quad \text{Result} = \text{Overlay}( \ \text{Antecedent}, \ \text{Intersection}(V, \ \text{Consequent}))
\]
else
  There is no result

This is simply the reverse of the ForwardTransition. It is incorrect, however. For our example rule and the condition vector \((B, T)\), the reverse transition would yield the condition vector \((A, M, T)\). Applying the forward transition to this vector yields \((A, N, T)\), which is not a subset of \((B, T)\). The problem is that the vector \((N, T)\) is a superset of the states that this rule can produce. The consequent \((N, T)\) can only be overlaid on states that match the antecedent \((A, M)\), which is much smaller than the universe of all states. Because of the restriction of the antecedent, this rule can produce the set \((A, N, T)\) only. We account for this by modifying the function as follows:

\[
\text{Antecedent} = R.lhs;
\text{Consequent} = \text{Overlay}(R.rhs, \text{Antecedent});
\text{if } V \text{ intersects Consequent then}
\text{Result} = \text{Overlay}(\text{Antecedent}, \text{Intersection}(V, \text{Consequent}))
\text{else}
\text{There is no result}
\]

Using this rule, the condition vector \((B, T)\) has no reverse transition because there it does not intersect the fully specified consequent. We next try to compute the reverse transition of \((A, T)\). This does have an intersection and will produce the condition vector \((A, M, T)\). If the rule is applied to \((A, M, T)\), the result is \((A, N, T)\), which is a subset of \((A, T)\).

There is still a problem, however. If the rule is applied to any subset of \((A, T)\), the result is \((A, N, T)\), which is a subset of \((A, T)\) and therefore correct. Our formulation of ReverseTransition will produce acceptable results but not all acceptable results. The problem is illustrated by the condition \(T\) in the consequent. \(T\) is not in conflict with any condition in the antecedent. This means that all conditions from the field of \(T\) are acceptable because they will all be overridden by \(T\) when applying the rule.

We need to define a function called \(\text{OverlayInverse}\). The \(\text{OverlayInverse}(E, F)\) will return the most general condition vector \(V\) such that \(\text{Overlay}(E, V) = F\). This is easily computed by removing from \(F\) all conditions found in \(E\). If \(E\) does not intersect \(F\), then there is no inverse because \(F\) could never be the result of an overlay by \(E\). The vector \(V\) defines all possible states such that an assertion of \(E\) will lead to the state space defined by \(F\). Using the \(\text{OverlayInverse}\) function, we can now correctly define ReverseTransition:

\[
\text{ReverseTransition}(V, R)\]
\[
(\text{Antecedent} = R.lhs;
\text{Consequent} = \text{Overlay}(R.rhs, \text{Antecedent});
\text{if } V \text{ intersects Consequent then}
\]
\begin{verbatim}
Result = Intersect( Antecedent,
              OverlayInverse( Consequent, V ) )
else
    There is no result
}

5.5. Detecting Conflict

Two models of rule conflict were defined in Section 4—the weak-conflict model, which says that two rules are in conflict when there exists a state for which each rule asserts some conflicting condition, and the strong-conflict model, which says that two rules are in conflict if there exists a state in which both rules can fire. The weak-conflict condition is the minimal condition that will allow a PPS to function.

**Strong-Conflict Detection.** To detect strong conflict in a rule set, we compare each rule \( R \) to every other rule \( S \) in the set:

\[
\text{StrongConflict}(R,S) = \text{Intersects}((R.\downarrow\text{-}ue, R.\text{lhs}), (S.\downarrow\text{-}ue, S.\text{lhs}))
\]

If the intersection exists, then, for those states in the intersection of the left-hand sides of each rule, both rules could fire.

**Weak-Conflict Detection.** To detect weak conflict, we similarly compared all rules \( R \) to every other rule \( S \):

\[
\begin{align*}
\text{SharedState} &= \text{Intersection}(R.\text{lhs}, S.\text{lhs}) \\
R0 &= \text{Overlay}(R.\text{rhs}, \text{SharedState}) \text{ result of applying } R \text{ to shared state} \\
S0 &= \text{Overlay}(S.\text{rhs}, \text{SharedState}) \text{ result of applying } S \text{ to shared state} \\
\text{WeakConflict}(R,S) &= \text{not Intersect}(R0,S0) \text{ and } (R.\downarrow\text{-}ue = S.\downarrow\text{-}ue)
\end{align*}
\]

If applying rules \( R \) and \( S \) to the states that both rules have in common will produce states that have no intersection, then there is no possible interpretation of both rules that will yield a consistent state. This is the basic requirement for interpreting the PPS. The original PPS interpretation accommodated weak conflict by rule blocking at run time. This allowed for default rules with general capability that could be overridden by more specialized rules.

5.6. Special Sets of States

There are several special sets of states that can be computed that will yield useful information about the nature of the dialogue specification.

**Dead States.** As described earlier, a dead state is one on which no rule can fire. To compute the set of dead states, we start with the universal set (which is a condition vector with "don't cares" on all fields) and then
remove the sets covered by the left-hand side of each rule. This is accomplished using the difference operator. The resulting list of condition vectors describes those states in which no rule can fire. Simplification of the resulting vector list is important so that the results can be clearly presented to the dialogue designer. Analysis of the set of dead states by a dialogue designer can quickly detect dialogue design problems that are not readily obvious by manually reading the rules.

**Producible States.** The producible states are the set of all states that can be produced by some rule. This set is constructed by forming a condition vector list from the right-hand sides or consequences of each rule and then simplifying the list. Any state that is not in this set can never be generated and therefore is unreachable. There might be states that are producible but still cannot be reached from the start state because there is no set of rules that would make the transition possible. The set of producible states, however, is a cheap upper bound on the set of reachable states. Taking the complement of the producible states, simplifying it, and presenting it to a dialogue author can help in detecting cases that have not been accounted for.

Notice that user events are not important in computing the set of dead states or the set of producible states. We assume that any input condition can be generated by a user.

**Producible Dead States.** A most interesting set of states is the producible dead states. This set is formed by intersecting the set of dead states with the set of producible states. It is very important to determine whether any of the producible dead states can be reached by some sequence of inputs. This set is preferred over the set of dead states because it is usually smaller and frequently empty. If this set is empty, then the user can never get trapped where there is no possible input.

Presenting the set of producible dead states to a dialogue designer is also very helpful in highlighting potential difficulties with the dialogue. If this set is not empty, it is also helpful to highlight for the designer those rules the right-hand side of which can produce a dead state. This clarifies for the designer exactly where the problems might arise. This information will help answer the question of whether a given specification can deadlock (Q7, Figure 4).

5.7. State Reachability Algorithms

The questions posed in Figure 4 are all variations on state reachability. The essence of this question is whether some sequence of inputs can lead the dialogue from one state or set of states to another. If such is possible, the nature of the input sequence required is also of interest.
Special Issues in State Graphs

A state space definition and a set of rules together form a directed graph on which the states are the nodes and the rules define transitions between states. There are well-known graph algorithms for determining which nodes of a graph are reachable from other nodes. There are, however, several problems with traditional graph searching algorithms. The first is that a PPS defines an exponential number of states. Representing a single state from an exponential set is not a problem. The set of integers between 0 and $2^N$ is exponential, but its representation requires only $N$ bits. Enumerating all the states in an exponential set is, however, a serious problem. Enumerating the transitions between an exponential set of states is even more daunting. It is exactly these problems with state machines that the PPS was designed to alleviate. A second problem is that we frequently want to define a set of start states and a set of target states in determining reachability. For example, given the start state of the dialogue, can we reach any members of the set of dead states? What we need, then, is algorithms that work in terms of sets of states rather than individual states.

In considering these algorithms, it helps to remember their purpose. In many cases, we are not interested so much in theoretical reachability as in reasonable reachability. If it takes 50 inputs to reach one state from another, it might be theoretically reachable but not reasonably so. This means that we can accomplish most of our purposes by placing a limit on the number of inputs in a reasonable sequence. This will seriously constrain the algorithms and prevent several problems. It should be noted, however, that reasonable reachability has important limitations. If, for example, one uses limited reachability to show that a user can never reach a dead state from the start state, such reasoning is seriously flawed. The assertion "There does not exist a path" is very different from "There does not exist a reasonably short path."

A particular problem in graph searching algorithms is the existence of cycles. Depth-first (backtracking) and heuristic graph search algorithms can get caught in an infinite loop if there are cycles in the graph. Breadth-first algorithms will always terminate if there is a path. If there is not a path, the breadth-first algorithms will also loop forever. In all these cases, the cycle problem is handled through various marking algorithms. The problem is that such marking algorithms require an enumeration of the state space, which we very much want to avoid. Imposing a limit on the reasonable length of a path will not prevent the exploration of irrelevant cycles but will prevent infinite loops in the algorithms.

We can also look at the kinds of input sequences that are reasonable. A most interesting set of paths are those that are monotonic in the way that conditions are asserted. The essence of monotonic paths is that we assume
that there is a relatively direct path to a given goal. Suppose, for example, that we want to reach the set of states characterized by the vector \((A, M)\). A monotonic path to \((A, M)\) would not use any rules that assert conditions that conflict with either \(A\) or \(M\).

In general propositional reasoning algorithms, the monotonicity assumption is usually not appropriate because it assumes that all problems have a relatively straightforward solution. We use monotonicity as a heuristic to control the explosion of our searches. In analyzing dialogue specifications, we should assume that the common case is the simple one because convoluted input sequences are an indication of bad dialogue design. Situations for which there is not a straightforward path to the goal will be confusing to users. If it is hard for a computer with large amounts of short-term memory to search out a path, then it will be almost impossible for users with limited short-term memory to do so. By assuming monotonicity, we might miss some paths. However, telling a designer that no acceptable path exists when, in actuality, a very convoluted path does exists does not jeopardize the goals of our analysis algorithms.

We can define two kinds of monotonicity—strictly monotonic and weakly monotonic. Each of these is discussed in turn.

**Strictly Monotonic Paths.** An input path is strictly monotonic if no condition, after being asserted, is ever contradicted along the path to the goal. To compute strict monotonicity, we must keep track of which conditions are given in the starting specification and which are asserted by some transition. Let us assume that, for each condition vector, there exists a function, \(\text{AssertedConditions}\), that will return only those conditions that have been asserted by some rule. The \(\text{AssertedConditions}\) function would remove all conditions that are being carried along from the initial conditions. This would mean that condition vectors would have to carry this additional information with them.

If we have an \(\text{AssertedConditions}\) function, then we can determine if a rule \(R\) is strictly monotonic with respect to a condition vector \(C\):

\[
\begin{align*}
\text{StrictlyMonotonic}(R, C) = \text{Intersects}(R.lhs, C) & \quad \text{Rule can be applied to } C \\
& \quad \text{and} \\
& \quad \text{Intersects( Intersection}(R.lhs, \text{AssertedConditions}(C)), R.rhs) \quad \text{Check for conflict}
\end{align*}
\]

If all the rules in a path are strictly monotonic, then the path is strictly monotonic, and the user will have not taken any side trips along the way. Many of our algorithms are general in that they do not require strictly monotonic paths. It is, however, a helpful heuristic to give strictly monotonic rules priority over other rules when searching for an input path.

It is important to note that strict monotonicity can prevent searching cycles but does not necessarily lead toward the target state set. In fact, after
a strictly monotonic sequence asserts a condition that conflicts with the target state set, the target state set can never be reached by a strictly monotonic path. This is helpful in pruning the search for paths to the target.

Weakly Monotonic Paths. It is not always possible for a path to be strictly monotonic. Suppose that one has a dialogue for which there is a property sheet that defines search criteria. If we are in a state in which no search criteria have been defined, and we want to perform a search, then we first need to assert the conditions that bring up the property sheet to specify the search; we then need to contradict those conditions to take down the property sheet before the search can be performed. Without bringing up the property sheet, we cannot perform the search, but the process of bringing up and putting down the property sheet is not strictly monotonic.

In our counterexample for strictly monotonic rules, we assumed that there were conditions that controlled the visibility of the property sheet. These conditions were required in order to achieve the subgoal of specifying search criteria, but they were not directly related to the goal of performing a search, which was dependent solely on the existence of search criteria. We can accommodate this situation by adopting a model of monotonicity, which considers only the conditions in the target state and ignores any nonmonotonicity in other fields.

In weak monotonicity, we can use a variant of Hamming distance as a heuristic for selecting rules. Our distance function between two condition vectors is defined as follows:

\[
\text{Distance}(\text{Target}, \text{Current}) = \text{NumberOfConditions}(\text{Target}) - \text{number of conditions from Target that are found in Current}
\]

If the distance between Target and Current is zero, then Current is a subset of Target, and the target state set has been reached.

Weak monotonicity for a given rule \( R \), a current condition vector \( C \), and a target condition vector \( T \) is given by:

\[
\text{WeaklyMonotonic}(R, C, T) = \begin{cases} 
\text{Intersects( } R.\text{lhs, C ) Rule can be applied} \\
\text{and } \\
\text{Distance}(T, C) \geq \text{Distance}(T, \text{ForwardTransition}(C, R)) 
\end{cases}
\]

In essence, a rule is weakly monotonic if it does not generate a set of states that are further from the goal than the original set. The rules that raise and lower a property sheet would leave the distance to the search goal unchanged, because they do not involve any of the conditions of the target state set.
A useful heuristic, then, is to follow first those rules that yield the smallest distance to the target. In cases of a tie on distance between rules, we can give priority to rules that are strictly monotonic. We still have the option of following rules that are not strictly monotonic in a general algorithm, but we can use monotonicity to focus our search and prevent useless wandering in most cases.

Requiring that all paths be strictly or even weakly monotonic is sometimes too restrictive. Suppose that our dialogue has a selection mode and an edit mode. Suppose also that, in order to delete an object, there must be a selected object, and we must be in edit mode. If we are in edit mode, and there is no object selected, then there is no monotonic path that will allow us to delete an object, because we would first need to leave edit mode and enter selection mode. This transition is not even weakly monotonic. We would then select an object, return to edit mode, and then delete it. In thinking about cases in which nonmonotonic paths are required, they most often are a result of modality in the dialogue. We can use monotonicity as a heuristic, but we must allow for the general graph searching case. It can, however, be helpful to keep track of how many nonmonotonic rules of each type were required to discover a path. This information can help designers in detecting convoluted dialogue or excessive modality.

**Simple State Reachability**

The easiest reachability analysis is to ask if a target state \( T \) can be reached from some start state \( S \). If we begin with a full state specification, every rule will generate a fully instantiated state. No more general specifications will be created. This means that, although a general algorithm is described that can work with sets of states, the complete state space might be searched anyway. The solution to this problem is to use the heuristics described earlier.

The algorithm is driven by a queue that contains the following information in each entry:

- **Current**: State set to be examined
- **NSteps**: Number of rules that have been applied to reach this state set
- **NNonWeak**: Number of rule transitions that were not weakly monotonic
- **NNonStrict**: Number of rule transitions that were not strictly monotonic
- **ProducingRule**: Identifier for the rule that produced the current state set
- **Previous**: Pointer to queue entry from which this one was produced

This queue is a priority queue that sorts the entries. Changing the sort order will change the heuristic properties of the search. This ordering is discussed later.

Remember that, when working with rules, this algorithm ignores all inputs and semantic actions. The algorithm is trying to explore what is
Figure 5. Reachability algorithm.

Reachable(S,T)
{
    Entry.Current = S;
    Entry.NSteps = 0;
    Entry.NNonWeak = 0;
    Entry.NNonStrict = 0;
    Entry.ProducingRule = NULL;
    Entry.Previous = NULL;
    Queue.Enter(Entry);
    Done = False;
    While not Done Do
    {
        OldEntry = Queue.Remove();
        If OldEntry.Current == T
        {
            Success(OldEntry);
            Done = True;
        }
        If (OldEntry.NSteps < MaximumUsableSteps)
        {
            For each rule R
            {
                If (R.lhs intersects OldEntry.Current)
                {
                    NewEntry.Current =
                    ForwardTransition(OldEntry.Current, R);
                    NewEntry.NSteps = OldEntry.NSteps+1;
                    If (WeaklyMonotonic(R,OldEntry.Current,T))
                    {
                        NewEntry.NNonWeak = OldEntry.NNonWeak;
                        If (StrictlyMonotonic(R, OldEntry.Current))
                        {
                            NewEntry.NNonStrict = OldEntry.NNonStrict;
                        }
                        Else
                        {
                            NewEntry.NNonStrict = OldEntry.NNonStrict+1;
                        }
                    }
                    Else
                    {
                        NewEntry.NNonWeak = OldEntry.NNonWeak+1;
                        NewEntry.NNonStrict = OldEntry.NNonStrict + 1;
                    }
                    NewEntry.ProducingRule = R;
                    NewEntry.Previous = OldEntry;
                }
            }
            If (Queue.Empty())
            {
                Done = True;
            }
        }
    }
}

Note. See text for explanation.

possible for a given rule set. The algorithm is presented in Figure 5 for a given target state T and start state S.

The algorithm in Figure 5 is guaranteed to find a path from S to T, if one exists and provided that MaximumUsableSteps is set to the maximum number of possible states. The problem with such a large number is that it is possible that the entire state space will be enumerated.

Even with the MaximumUsableSteps limit, it is still possible that the entire state space will be explored. Consider the example of N inputs,
which should be received in any order provided each input is received exactly once. There is no path through such a dialogue that is longer than $N$ steps; however, there is an exponential number of such paths. To resolve this, we need to consider the sorting order of the priority queue.

If we sort the queue entries in ascending order on $N_{\text{Steps}}$, we get a breadth-first search of the state space that, in our $N$-inputs example, is exponential. If we sort the queue in descending order on number of steps, we get a depth-first search that, in the $N$-inputs example, finds a path in $N$ steps. The problem with depth-first search is that there might be cycles or other ways to wander off into useless paths. The $N$-inputs problem is special in that all legal paths lead to the target state. If a depth-first search finds its way into a path that is not productive, it can waste a large amount of time and explore a large number of unnecessary states. The following three sort keys yield the kind of search that we want:

- $N_{\text{NonStrict}}$: Ascending
- $N_{\text{NonWeak}}$: Ascending
- $N_{\text{Steps}}$: Descending

The $N_{\text{NonStrict}}$ field is the primary key, with $N_{\text{Steps}}$ being the least important key. $N_{\text{NonStrict}}$ and $N_{\text{NonWeak}}$ count the number of times that a path has moved away from the goal. By making them the primary keys, the search focuses on paths that lead most directly to the target. Given paths that are monotonic or almost monotonic, the descending sort on $N_{\text{Steps}}$ will perform a depth-first search. There are other sort orders that can deal with special situations more profitably, but this model will perform very well on dialogues that do not have lots of modes. This sort order will still find a path if one exists within the MaximumSteps constraint. In fact, it will find the path that passes through the fewest number of different modes.

**Input Path Analysis**

Because each entry keeps track of the rule that produced it and a pointer to the entry from which it was generated, we can reconstruct the path through the dialogue discovered by this algorithm. There are several things that can be done with this information. The path can be presented to the designer for evaluation. The length of the path can be reported. If there are known costs for using various inputs (e.g., the cost of keyboard vs. mouse), the cost of the path can be reported. The algorithm can be modified to sum the input costs and to give priority to lower cost paths. This would be done by summing costs rather than counting steps.

A variation on path costing, proposed by Bleser and Foley (1982) and used by Olsen and Halversen (1988), evaluates the cost of switching input
devices. A matrix of costs can be used. For example, a mouse input followed by another mouse input is cheaper than a mouse input followed by a keyboard input, which would require moving the hands. Such a cost matrix could be used to discover minimal cost paths through the dialogue. The cost to be summed into each entry would use the input of the current rule and the input of the previous rule to determine the cost associated with a given transition. This information can all be computed and presented to a designer for study.

**State Set Reachability**

Other than input path analysis, the testing of whether one particular state can be reached from another is not very interesting. As was pointed out in Section 4, the interesting questions concern the reachability of sets of states corresponding to the preconditions for a rule. Q1 (Figure 4) asks whether the set of states corresponding to the preconditions of a rule can be reached from the initial start state. Q2, Q3, and Q4 ask the same question but with a set of start states.

Q6 is similar. Suppose that a rule exists that will assert a condition to bring up the help system. The left-hand side of this rule defines the set of conditions required to bring up the help system. Typically, these conditions define not a single state but the set of states that have the desired conditions. If we want to know for a given state how hard it is to get help, then we want the path from that state to any of the states from which help can be asserted. As the algorithm just shown uses the ForwardTransition function, it already works on state sets rather than just on states. The only change that must be made is in the test for success. This is changed to:

```python
If OldEntry.Current intersects T
  { Success( OldEntry );
    Done = True;
  }
```

This minor change will allow us to explore sets of states rather than individual states.

**Dead State Reachability**

Q7 (Figure 4) is to determine if it is possible to reach a dead state from the start state. A short form analysis is to first compute the producible dead states. If this set is empty, then there is obviously no way to reach a dead state. If the set of producible dead states is not empty, then the question is still hanging as to whether or not a user could fall into a dead state. There are three possible methods for checking this condition—using state reachability, computing the set of all states reachable from the start state, and using reverse reachability from the dead states.
Using State Reachability. The first approach is to take each condition vector in the set of producible dead states and check for its reachability from the start state. The problem is what to use as a limit on number of steps. Remember that, without such a limit, the state reachability algorithm might not halt. In particular, it will not halt if there is no path. This is particularly disconcerting because that is exactly the case that we want. One possibility is to set the number of steps to be slightly larger than the number of rules. This is based on the idea that it is probably not useful to apply a rule more than once. This limit, however, is a pragmatic suggestion rather than a true solution. With such a limit, this approach does not guarantee that no such path exists. This approach will, however, quickly point out the most likely problems. However, there is still the problem that, after long usage, the user might fall into a dead state.

Computing the Set of All States Reachable From the Start State. Another approach is to compute the set of all reachable states. For this, we use a fixed-point algorithm that steadily grows the set of reachable states until no change occurs:

```plaintext
ReachableStates = StartState;
NewlyReachedStates = StartState;
While not Empty(NewlyReachedStates)
{
    C = some condition vector removed from NewlyReachedStates;
    For each rule R
        Reached = ForwardTransition(C, R);
        NewlyReachedStates = Union(Difference(Reached, ReachableStates),
                                    NewlyReachedStates);
    ReachableStates = Union(Reached, ReachableStates);
}
```

Given a finite state space, this algorithm is guaranteed to halt and to compute the exact solution. Its problem lies in the computation of Union. Because StartState is a fully instantiated state, ForwardTransition will return a fully instantiated state. If Union is naively implemented as a list of all condition vectors in the set, then the entire reachable state space will be enumerated in ReachableStates. This is entirely impractical. If, however, Union contains logic for merging condition vectors into more general condition vectors in which it is valid, then the algorithm will not require full enumeration of the state space and will also move much more quickly because sets of states can be processed at each iteration instead of at individual states.

Having computed the set of reachable states, we can intersect this with the producible dead states to locate reachable dead states.
Using Reverse Reachability From the Dead States. One of the problems with the algorithm of reachable states, however, is that ForwardTransition on any condition vector will always produce a condition vector that has the same number of or fewer states. This means that ForwardTransition is always working against our attempts at generalization of condition vectors to conserve time and space. We can overcome this by using the ReverseTransition function, which will always produce a condition vector for an equal or larger set. What we will do is start from the dead states and work backward to see if we can reach the start state:

```plaintext
ReachableStates = the dead states;
NewlyReachedStates = the producible dead states;
While not Empty(NewlyReachedStates)
    { C = some condition vector removed from NewlyReachedStates;
    For each rule R
        { Reached = ReverseTransition(C, R);
        NewlyReachedStates =
            Union(Difference(Reached, ReachableStates),
                  NewlyReachedStates);
        ReachableStates = Union(Reached, ReachableStates);
        If NewlyReachStates contains the start state
            { Stop and report the problem }
        }
    }
```

By starting with sets of states and working backward through ReverseTransition, we will be working with condition vectors that describe large sets of states that will generalize much more quickly without enumerating all the individual states.

5.8. Maximum Minimal Paths to Reach Any Given Rule

Q5 (Figure 4) was explored in Section 5.6 as a question of reachability. Another way of formulating this question is the maximum number of inputs required to make some rule available. Again, we can take the example of help. Suppose that this is initiated by a particular rule. A useful question to ask would be “Where in the dialogue is the most difficult place to get help, and how hard is it?” This question can be addressed by working backward from the desired preconditions to see what states can be reached and how many steps are required. This is a variation on the algorithm of reverse reachability. This algorithm can also make use of a limit on the number of steps taken. We might want to ask “What states require more than 4 inputs before receiving help?” This will show us exactly where getting help (or any other semantic condition) is convoluted for the user and at the same time will eliminate costly computation in working through that convolution:
StepsTaken = 0;
ReachedStates = the antecedents of any rules which assert the desired condition;
LastReachedStates = ReachedStates;
While not Empty(LastReachedStates) and StepsTaken <= MaxSteps
{ StepsTaken = StepsTaken+1;
  NewlyReachedStates = empty;
  For each condition vector C in LastReachedStates
  { For each rule R
    { Reached = ReverseTransition(C, R);
      NewlyReachedStates =
      Union(Difference(Reached, ReachedStates),
            NewlyReachedStates);
      ReachedStates = Union(Reached, ReachedStates);
    }
  }
  LastReachedStates = NewlyReachedStates;
}

When this algorithm stops, StepsTaken will contain the number of steps taken by the algorithm. If this is less than or equal to MaxSteps, then this will contain the maximum number of inputs required to reach the desired semantic conditions. If StepsTaken is greater than MaxSteps, then the difference between the set of states reachable from the start state and ReachedStates will yield those states at which it requires more than MaxSteps to assert the desired conditions. These can be presented to the designer as problem states.

5.9. Rule Set Connectedness

Q6 (Figure 4) addresses the relation between rules and situations more global than simple state reachability. This involves whether there are disjoint sets of rules that are not reachable from one another. Such disconnectedness might be a positive or a negative attribute, depending on the purpose of the dialogue.

In a normal application, one would not like to have a set of states that are reachable but that the user cannot escape. This problem arises when the set of states that is reachable from the consequence of some rule is significantly smaller than the set of reachable states. It might be possible to get into some dialogue in which rules can fire, but none of them can leave that dialogue. This would be inappropriate in most cases.

In some cases, such disconnectedness is very desirable. Suppose an application has a log-in procedure with some security protection (e.g., a password). One would like to determine exactly which rules can ever be reached from the states that have the InvalidPassword condition. In this case, we want the InvalidPassword condition to be isolated from rules that perform the work of the application. In actuality, this case is slightly more complicated. If the user enters an invalid password, we do
want to be able to reenter the password or pass through the log-in procedure again to ultimately reach the meat of the application. Our restriction is not to prevent access but to ensure that such access must pass through the prescribed process. A similar case would arise on quitting, when one would want to ensure that the program cannot quit without passing through the shutdown dialogue, which closes or discards any open files.

**Simple Connectedness.** The issue of simple connectedness can be addressed by knowing the set of states that can reach the antecedent or left-hand side of each rule. This is a variant on the algorithm of reverse reachability from dead states. Instead of initializing `ReachableStates` with the dead states, we initialize it with the left-hand side of some rule. Then the algorithm will yield all states that can reach that rule. If we subtract this from the set of producible or reachable states, we are left with the reachable states that cannot reach this rule. In most cases, this should be the empty set. For a complete analysis of the dialogue, this approach can be applied to every rule.

**Restricted Connectedness.** Good dialogues are always completely connected. In the case of the password or quit dialogues described earlier, we want the connectedness to pass through specific dialogue fragments only, such as the log-in or shutdown dialogues. To test for such restricted connectedness, we first remove from the rule list those dialogue fragments that constitute the required connection and then run the analysis of simple connectedness. If the specified dialogue fragments are truly required to pass from one segment of the state space to another, this analysis will show those segments to be disconnected. If there is a way around the specified fragment, then the segments will still be fully connected.

6. CONCLUSIONS

The requirements for any notation to be usable for user interface specification are discussed in the introduction to this article (see Curry & Monk, in press, for a more detailed discussion of this issue). Three of these requirements have been most influential in shaping the approach described here—ease of learning, animation, and automated analysis.

**Ease of Learning.** We have no empirical evidence that our notation is easy to learn, as thus far it has only been used within the research context in which it was developed. Readers must decide for themselves, bearing in mind the alternatives. For example, it is difficult to imagine explaining Z or VDM to a nonmathematical audience in the few pages used here to explain PPS. Tutorial material describing the methods used here is under development. These materials will be empirically evaluated.
Animation. A major problem facing the designers of modern GUIs is matching the high-level behavior of the user interface to the work tasks it is to support. An important part of this problem is the general inadequacy of any static specification in conveying the dynamic properties of the software described. For this reason, facilities that allow a designer to animate a specification are most important. This is one of the most important strengths of PPS. Animation provides a way of evaluating a high-level dialogue model against a task model.

Automated Analysis. The main justification for using a formal notation is that it permits analysis and proof. However, few designers have both the skills and the time required to do this. For this reason, any formal notation for use in a real design setting must come with automated routines for analysis. This article has focused on analysis of PPS specifications. Several design questions amenable to automatic analysis have been suggested along with practical algorithms that can be used to perform them.

The next step in this research is to build a tool to do these analyses. The practical experience of using such a tool is necessary to determine which analyses are most useful and how they should be applied within the design method sketched in Section 2. Global checks for rule conflict, deadlock, and rule set connectedness could be performed whenever the designer thinks the specification is reasonably complete. Similarly, it would be possible to check for reversibility of effect and mode for each rule in the rule set. In the former case, this would require that, for each rule, the designer should specify which rule has a side effect that reverses the rule's own side effect. This might be a useful design exercise in itself.

The other reachability questions—task completeness, task connectedness, and accessibility—require the designer to identify rules that are critical in terms of the task model. Accessibility is to be assessed for any rule that supports a task that could be performed at many points during the user's work. Getting help and saving and printing a document are used as examples. Similarly, task completeness and connectedness require the designer to identify rules associated with crucial tasks from the task model. How easy this is to do in practice and what guidelines can be provided for designers can be determined only by experience. However, it is not unreasonable to claim that PPS, along with the algorithms specified here, can be very helpful in user interface design.

Algorithm Issues. Although the state machine model provides a firm algorithmic base for our analysis, the PPS extension to state machines generates an exponential number of states and paths to traverse. We solve
this problem in three ways. First, we use algorithms that work on sets of states rather than on individual states. Second, we recognize that excessively long input paths and complicated input paths indicate bad user interface design and therefore can be counted as failures without completely traversing those avenues. This allows us to put practical limits on our algorithms, which otherwise would be theoretically unjustified. Third, we design several algorithms that work backward from the target. This provides a payoff in that each step produces more general descriptions with concise machine representations rather than ever more specific, fragmented, and space-consuming representations.

NOTES

Support. This work was partly supported by National Science Foundation Grant IRI-9123468 to Dan R. Olsen, Jr. and by DTI/SERC Grant 1708 to Andrew F. Monk.

Authors’ Present Addresses. Dan R. Olsen Jr., Department of Computer Science, Brigham Young University, Provo, UT 84602. E-mail: olsen@cs.byu.edu; Andrew F. Monk, Department of Psychology, University of York, Heslington, York, YO1 5DD, England. E-mail: am1@tower.york.ac.uk; Martin B. Curry, Sowerby Research Centre, British Aerospace plc, FPC 267, Filton, Bristol, BS12 7QW, England. E-mail: curry@src.bae.co.uk.

HCI Editorial Record. First manuscript received August 23, 1993. Revision received March 3, 1994. Accepted by Richard Young. Final manuscript received July 25, 1994. – Editor

REFERENCES


