Using Harel’s Statecharts to Model Business Workflows

Wai Yin Mok, University of Alabama in Huntsville, USA
David Paper, Utah State University, USA

ABSTRACT

In this paper, we model business workflows using Harel’s statecharts. Mapping to statecharts allows us to systematically identify potential workflow problems. It also allows us to investigate specific properties inherent in actual business workflows. Our research focuses on three desirable properties of active database systems — termination, confluence, and observable determinism. For termination and confluence, we develop algorithms to provide a theoretical lens linking desirable active database system properties to workflow management systems problems. We initially validate our algorithms by mapping business workflows from a case study. Our research thus builds preliminary theory by developing a systematic method for identifying workflow problems.

Keywords: Business workflows, workflow management systems

INTRODUCTION

Business workflows can be well defined, predictable, and frequently executed. We refer to these as structured business workflows. Workflows with these characteristics can be automated by machines to reduce clerical tasks and potential human intervention errors. We can use workflow management systems (WMS) to facilitate automation of structured business workflows. WMS, which are new generations of computerized systems, are designed to manage automated parts of business workflows (Brunwin, 1994). By separating workflow definitions from application software, WMS provide process and knowledge.
independence, much like data independence provided by database management systems.

We propose using Harel’s statecharts to model structured business workflows (Harel, 1987) for three reasons. First, Harel’s statecharts are used in the Unified Modeling Language (UML) as a means for behavioral modeling (Object Management Group, 1999). Since the UML is the standard modeling language of the Object Management Group, Harel’s statecharts will soon become common. Second, statecharts are easy to understand and they do not have the problem of exponential growth of states that plague ordinary state transition diagrams (Harel, 1988). We shall elaborate this point in the Related Work section. Third, their semantics are rigorous enough for formal analysis on various aspects of structured business workflows (Harel and Naamad, 1996).

In the framework of statecharts, we will show how to model workflow concepts and present algorithms that determine whether a given business workflow has certain predefined properties. We will then use a case study with Moore BCS to explore the characteristics of a business workflow. The algorithms we develop in this study will become part of a software design tool that we will develop in the future.

RELATED WORK


Active database systems have been studied extensively (Paton and Diaz, 1999). Active database systems and workflow management systems are related since both types of systems employ triggers to respond to external and internal events and exceptions. We are interested in three important properties of active database systems in this paper, namely termination, confluence, and observable determinism, which are formally defined in Allen, Hellerstein, and Widom (1995). More discussion on active database systems, which include several research prototypes and commercial products, can be found in Zaniolo (1997).

The statemate approach, which uses statecharts in modeling reactive systems, is described in Harel and Politi (1998) and its semantics in Harel and Naamad (1996). By far, the statemate semantics of statecharts is the most rigorous and precise execution semantics defined for statecharts and it has been in use for more than ten years (Harel and Naamad, 1996). Here we point out the most significant aspects of the execution semantics. The reader may consult Harel and Politi (1998) and Harel and Naamad (1996) for details.

The behavior of a system described in statemate semantics is a set of possible runs, each representing the responses of the system to a sequence of external stimuli generated by its environment. A run consists of a series of detailed snapshots of the system’s situation; such a snapshot is called a status. The first in the sequence is the initial status, and each subsequent one is obtained from its predecessor by executing a step (see Figure 1).
Some of the general principles of state machine semantics are as follows:

1. Reactions to external and internal events, and changes that occur in a step, can be sensed only after completion of the step.
2. Events are “live” for the duration of one step only, the one following that in which they occur, and are not “remembered” in subsequent steps.
3. Calculations in one step are based on the situation at the beginning of the step (e.g., the states the system was in, the activities that were active, and the values of conditions and data-items at that time). Updates of data items only occur at the end of a step.
4. A maximal subset of non-conflicting transitions is always executed.

Item 3 above deserves more explanation. As an example, suppose there is an action: “X := X + 1; Y := X * 5;” which is executed in a step. Further suppose that X is equal to 4 at the beginning of the step. Because of Item 3, after executing the step, X becomes 5 and Y becomes 20. Note that every computation of the action does not influence any other computation of the action. The semicolon separating the actions means, “do this too” rather than “and then do” in state machine semantics.

Using activity diagrams to model workflows is discussed in Chapter 19 in Booch, Rumbaugh, and Jacobson (1999). Note that activity diagrams are special statecharts in which all of the state transitions are triggered by completion of activities in the source states. Activity diagrams are not designed to handle events. Since exceptions may happen during the execution of a workflow instance and exceptions are best modeled as events, activity diagrams can only model very simple workflows. In this sense, statecharts are more appropriate for modeling realistic workflows.

Before we show how to use statecharts to model workflow concepts, we present an example, adapted from Harel (1987), that shows the problem of exponential growth of states that plague ordinary state transition diagrams. In

![Figure 1: Status and Step](image1)

![Figure 2: Exponential Growth of States](image2)
Figure 2, a statechart and its equivalent state transition diagram are presented. Note that by making use of an and-state in the statechart, we can easily model concurrency in a system by orthogonal components in the and-state. On the other hand, to perform the same modeling in the equivalent state transition diagram, we require six states. Using the same reasoning, for an and-state with two orthogonal components with a thousand states in each of them, the equivalent state transition diagram would require a million states. It is easy to see that it is difficult, if not impossible, to model concurrency in ordinary state transition diagrams because of the problem of exponential growth of states.

**BASIC CONCEPTS AND TERMINOLOGY**

The Workflow Management Coalition (WFMC) has published numerous documents on various aspects of business workflows. We now introduce some basic concepts and terminology defined by WFMC. A *business workflow*, or simply a workflow, is a set of activities, which collectively realize a business objective. An insurance claims process is an example. A workflow is defined in a *workflow definition* that consists of a network of activities. Usually a workflow definition is a formal representation of a business workflow. An activity is a logical step within a workflow. As such, it is usually the smallest unit of work within a workflow. Further, an activity can be manual or automated. A *workflow management system* is used to manage automated activities, but not manual activities. A *workflow instance* is the representation of a single execution of a workflow. It has its own workflow instance data and is capable of independent control as it progresses towards completion. The processing of an insurance claim for a particular customer is thus an example of a workflow instance of the insurance claims process.

Similarly, an *activity instance* is the representation of a single invocation of an activity within a workflow instance. Several activity instances may be associated with a workflow instance, but one activity instance cannot be associated with more than one workflow instance.

**MODELING WORKFLOW CONCEPTS**

A business workflow can be formally represented by a statechart. Each workflow instance has its own copy of the statechart. An activity of a workflow, whether it is manual or automated, is represented by a state in the statechart. An activity is being carried out only if the system resides in the state that corresponds to that activity.

Transitions between activities are thus modeled as transitions between states in the statechart, which are triggered by *events* and guarded by *conditions*. Events can be external (generated by elements outside the statechart) or internal (generated by elements inside the statechart). A transition between a source state and a target state will take place if and only if the system currently resides in the source state, and the event of the transition occurs, and the conditions that guard the transition are true. In other words, the system must be in the source state, the event must occur, and the conditions must be true for the transition to take place.
The four possible types of routing in workflows are sequential, parallel, conditional, and iterative (van der Aalst, 1998). In the following subsections, we show how to model these four types of routing in statecharts, and illustrate several statechart concepts that are relevant in modeling workflows.

### Sequential Routing

Activities are executed one after the other in sequential routing. In Figure 3, E1 is the event, C1 is the condition, and A1 is the action of the transition between state A and state B. In Figure 3, the system may still reside in state A unless at the instant E1 occurs, C1 is true. Events are instantaneous. A transition between a source state and a target state will take place if and only if the system currently resides in the source state, and the event of the transition occurs, and the conditions that guard the transition are true. In other words, the system must be in the source state, the event must occur, and the conditions must be true for the transition to take place.

### Parallel Routing

In contrast to sequential routing, activities can be executed concurrently in parallel routing. This is exactly why several activity instances may associate with a workflow instance. We do not specify conditions in Figure 4. By default, they are assumed to be the completion of the activities represented by the source states. If additional conditions are given for a transition, then the actual guarding condition of the transition is the conjunction of the additional conditions and the completion of the activity represented by the source state, unless otherwise stated. As an example, in Figure 3, the actual guarding condition of the transition from state A to state B is the completion of the activity represented by state A and the
additional condition C1. Events and actions are also omitted. If the event of a transition is omitted, then the system will check the condition continuously. Thus, whether the transition will take place may depend solely on the condition of the transition (Harel and Politi, 1998). Hence, transitions do not have to depend on any particular events and actions do not have to be performed during transitions. In Figure 4, E is an and-state, which has two orthogonal components. Being in E means being in these two components simultaneously. The fork construct specifies that when the system exits state A, it will enter states B and C simultaneously. The merge construct specifies that the system will leave state E only if it resides in states D and C simultaneously. However, because of the default conditions, this transition will only take place if activities D and C are both completed. Thus, synchronization occurs at merge constructs. Fork and merge constructs can be used to model the or-split and or-join defined by WfMC (Workflow Management Coalition, 1999).

**Iterative Routing**

Iterative routing is similar to conditional routing. Again, C1 and C2 are two mutually exclusive conditions. Whether or not the system stays in state B in Figure 6 depends on the truth or falsity of the iterative condition C1.

**Desirable Properties**

Workflow management systems and active database systems both employ triggers to respond to exceptions and events. Thus desirable properties of active

---

Additional condition C1. Events and actions are also omitted. If the event of a transition is omitted, then the system will check the condition continuously. Thus, whether the transition will take place may depend solely on the condition of the transition (Harel and Politi, 1998). Hence, transitions do not have to depend on any particular events and actions do not have to be performed during transitions. In Figure 4, E is an and-state, which has two orthogonal components. Being in E means being in these two components simultaneously. The fork construct specifies that when the system exits state A, it will enter states B and C simultaneously. The merge construct specifies that the system will leave state E only if it resides in states D and C simultaneously. However, because of the default conditions, this transition will only take place if activities D and C are both completed. Thus, synchronization occurs at merge constructs. Fork and merge constructs can be used to model the or-split and or-join defined by WfMC (Workflow Management Coalition, 1999).
database systems are also applicable to workflow management systems. We chose to examine three salient desirable properties of active database systems, namely, termination, confluence, and observable determinism. Given a statechart of a business workflow, we present several procedures to determine whether the given statechart has these properties.

Termination

As discussed in the Modeling Workflow Concepts section, external events are generated by elements outside the given statechart. Internal events, on the other hand, are generated by elements inside the statechart. Sometimes events can be generated both externally and internally. For example, consider the statechart of a machine. The event “power off” can be generated externally by an operator when he shuts down the machine or the event may be generated internally by the machine itself when it is overheated. Generation of events may lead to infinite execution of a statechart. In Figure 7, which is adapted from Figure 47 in Harel (1987), once the event E1 occurs externally, events E2, E3, E4, and E1 will be generated in this order internally, forever, meaning that the statechart will never terminate. A workflow design tool should be able to detect cycles of this sort before the actual deployment of the system. In this way, termination problems can be detected and corrected during modeling rather than in production.

There are certain distinguishing features in Figure 7. First, it leads to infinite execution and second, it contains cycles. In fact, a statechart that leads to infinite execution always has a cycle even though having a cycle in a statechart does not mean that the statechart will always lead to infinite execution. Third, in Figure 7, the transitions on the cycles are triggered by internally generated events or conditions that will never run out or never be false.

Note that in the third point above, the condition that the “internally generated events or conditions that will never run out or never be false” is important since internally generated events may eventually run out or conditions of transitions may eventually become false, as shown in the following example.

In Figure 8, if x is equal to 5 initially, we will only go through the loop 5 times. However, detecting transitions of this kind requires complicated analysis of the actions of the transitions. In the literature, Baralis, Ceri, and Paraboschi (1996) contain complicated algorithms for this kind of analysis, which may lead to a long execution time. Our techniques, on the other hand, are only based on reading the values of and writing values to data items.
Admittedly, our techniques do not have the precision of those in Baralis et al. (1996).

We now introduce Algorithm 1. Algorithm 1 provides a mathematical procedure that determines whether or not a given statechart terminates. The proof of Algorithm 1 theoretically validates the viability of termination as a critical property of business workflows.

**Algorithm 1**

**Input:** A statechart.

**Output:** Yes or no (Note that our analysis is very conservative in the sense that if our algorithm says “yes”, the statechart may still terminate because events may run out or conditions may become false on a transition. However, as we have just mentioned, detecting situations of this kind must be done by careful analysis of the actual computations of the transitions, which we do not perform here. As an example, in Figure 7, after examining the computation of the transition, we can conclude that the loop will eventually terminate. Nevertheless, for much more complicated transitions which may have hundreds or thousands of lines of code, analysis could be hard, if not impossible, to perform).

1. If there is no direct cycle constructed from states and transitions in the statechart, the statechart will terminate and we may stop and say “no”; otherwise let S be the set of such cycles. For each cycle s in S, if s contains a transition whose event can only be generated externally or whose condition can only be set to true externally, s will not cause non-termination and we may remove s from S.
2. If there is no cycle constructed from internally generated events or conditions in the statechart, the statechart will terminate and we may stop and say “no”; otherwise let E be the set of such cycles.
3. If there exists an element s in S and an element e in E such that the events that take the system from one state to the other in s is a subsequence of e, then the statechart will never terminate and we say “yes”; otherwise the statechart will terminate and we say “no”.

**Theorem 1:** Algorithm 1 specifies sufficient conditions for a statechart to terminate.

**Proof:** In Step 1, if there is no direct cycle constructed from the states and transitions in a statechart, the statechart will terminate since the activity associated with a state will terminate and each state in the statechart will only be visited once. In case there is such a cycle s, and there is a transition of s whose event can only be generated externally or whose condition can only be set to true externally, the completion of s depends on external interventions. Thus, s will eventually be stopped by external means. In Step 2, if there is no cycle constructed from internally generated events or conditions in a statechart, the events and conditions are not “self-feeding” which means the events will eventually run out and the conditions will eventually become false. On the other hand, in Step 3, if we can find such a cycle s and a self-feeding cycle e of events and conditions, then s will never stop once s is started.

As an example, in Figure 7, S is \{[A, B, A], [C, D, C]\} and E is \{[E1, E2, E3, E4, E1]\}. Let s be [A, B, A] and e be [E1, E2, E3, E4, E1]. The events that take the system from one state to the other in s is [E1, E3], which is a subsequence of e. Thus the statechart will never terminate.
In the Statechart Analysis section, we will use Algorithm 1 to demonstrate its ability to detect a non-termination problem from the actual workflow scenario illustrated in Figure 10. In the next section, we discuss confluence.

**Confluence**

Consider a set of non-prioritized transitions that are fired at the same time. If the final status of the system does not depend on their order of execution, then the system is **confluent**.

Whether a system is deterministic or not has a great impact on the confluence of the system. For example, in Figure 5, if C1 and C2 are not mutually exclusive, then both of them could be true at the same time and thus the system needs to nondeterministically choose either state B or state C to enter. To avoid situations like this, for each conditional routing, we require the user to prioritize the alternatives so that in case there is a tie, a tiebreaker is provided.

Two transitions are in conflict if there is some common state that would be exited if any one of them were to be taken (Harel and Naamad, 1996). Nonconfluent statecharts, or systems, are caused by nondeterminism of execution and conflicting transitions in the statecharts. In other words, when a statechart encounters nondeterminism (that is, when there is more than one possible execution sequence of the conflicting transitions in a step), the final database state may be different due to a different order of execution of the conflicting transitions. However, for some conflicting transitions, a different order of execution may still lead to the same final database state after the statechart becomes stabilized. This is because they do not have a read-write racing problem or a write-write racing problem, which are defined as follows:

Two transitions \( t_1 \) and \( t_2 \) have a **read-write racing problem** if \( t_1 \) reads the value of a data item \( x \) and \( t_2 \) writes a value to \( x \).

Two transitions \( t_1 \) and \( t_2 \) have a **write-write racing problem** if both \( t_1 \) and \( t_2 \) write values to a common data item \( x \).

The two racing problems mentioned above are related to concurrency control problems inherent in database management systems (Bhargava, 1999). In this paper, however, we adhere to the terminology used in the statechart literature. That is, we will keep using the terms read-write racing problems and write-write racing problems.

Figure 9 demonstrates a read-write racing situation. In this example, when the

---

**Figure 9. A Read-Write Racing Problem**

---

event “New Year” occurs, two transitions take place at the same time. Whether the new payment will be based on the new interest rate or the old interest rate depends on which transition is executed first. In this case, the statechart is not confluent.

We now introduce Algorithm 2 whose purpose is to identify a set of concurrently executed transitions that may lead to non-confluence.

**Algorithm 2**

**Input:** A set T of concurrently executed transitions.

**Output:** A partition of the transitions in T such that each partition class may lead to non-confluence. (Note that once again our analysis is very conservative in the sense that the set N outputted by Algorithm 2 may not lead to non-confluence. As in Algorithm 1, to conclude that there is non-confluence, we must carefully study the actual computations of the transitions, which we do not perform here).

1. We first create a graph G with each transition in T as a vertex in G. However, G has no edge in this stage.

2. If two distinct transitions t₁ and t₂ have a read-write racing problem or a write-write racing problem, we add an edge to the two corresponding vertices in G. We continue this step until no more edges can be added to G. Note that at most n(n-1)/2 edges are added to G if n is the number of transitions in T.

3. The transitions in each connected component of G with at least two transitions may lead to non-confluence.

**Theorem 2:** Algorithm 2 correctly identifies sets of transitions that potentially lead to non-confluence.

**Proof:** Note that Algorithm 2 partitions the set T into disjoint subsets of T. Consider a partition class C where C is a connected component in G. If C has at least two transitions, then a transition in C has either a read-write racing problem or a write-write racing problem with another transition in C. Switching the order of execution of these two transitions will cause different final states of the system.

For any such set N and for any transition in N, there is another transition in N such that they have either a read-write racing problem or a write-write racing problem. In this way, Algorithm 2 points out the set of transitions that may potentially lead to non-confluence to the analyst and the analyst may consult with the client to devise a solution to the problem.

The next theorem is interesting in the sense that the statemate semantics of statecharts avoid certain problems.

**Theorem 3:** If a statechart S implements the statemate semantics, then read-write racing problems will not cause non-confluence.

**Proof:** Since each transition is prioritized and calculations in one step are based on the situation at the beginning of the step and updates of data values only occur at the end of a step, the data values read during the execution of a step are all produced in the previous step. Thus any data values produced during the execution of a step will not be read by any transitions executed in the same step. Therefore, read-write racing problems will not cause non-confluence.

As an example, if the statechart in Figure 9 implements the statemate semantics, then the calculation of the payment will be based on the old interest rate rather than on the new interest rate. The calculation of the new interest rate, of course, is also based on the old interest rate.
rate. However, the new interest rate is not available to the other calculations that occur in the same step.

In the Statechart Analysis section, we will use Algorithm 2 to identify the set of transitions in Figure 10 that may lead to non-confluence.

**Observable Determinism**

A transition is *observable* if its action is visible to the environment. A good example would be “print Profit” where Profit is a variable. Consider a set of non-prioritized and observable transitions that are fired at the same time. If the order of the output of the system does not depend on their order of execution, then the set is *observably deterministic*.

**Theorem 4:** If a statechart $S$ implements the statemate semantics, then $S$ is observably deterministic.

**Proof:** In the statemate semantics of statecharts, all actual updates of data items (or variables) are done at the end of a step (Harel and Naamad, 1996). Thus, any values that are displayed or printed out during the execution of a step are updated at the end of the previous step. Hence, displaying values to the environment cannot interleave with updating of values within the system. Therefore, if a system implements the statemate semantics, then it is observably deterministic.

The next theorem shows the relationship between the properties Observable Determinism and Confluence.

**Theorem 5:** If a statechart $S$ is observably deterministic, then $S$ is confluent.

**Proof:** For each transition in $S$, we add the action “show the entire current status of $S$”. Thus, if $S$ is observably deterministic, then the printout of the status of $S$ will be deterministic, which means $S$ is confluent.

By Theorem 5, if a statechart is not confluent and its outputs are visible, then it is not observably deterministic. In the Statechart Analysis section, we will identify the set of transitions in Figure 10 that may lead to non-observable determinism.

**CASE STUDY**

A case study is introduced to theoretically validate our algorithms in a real-life context. Benbasat, Goldstein and Mead (1987) and Yin (1994) endorse the use of case studies to capture knowledge from practice. Our study develops theory concerning algorithms that prove the value of statecharts in determining termination, confluence, and observable determinism in workflows. However, we wish to extend theoretical development by testing these properties in a real-life context. The case study approach offers a vehicle to construct applied theory from scholarly theory.

Moore Business Communication Services (BCS) is located in Logan, UT. Moore BCS is a large company with approximately $2.45 billion in 1999 revenue. Moore BCS helps large corporations increase their competitiveness by improving the effectiveness of important business-to-customer communications. It provides consulting, project management, reengineering and distribution of high volume, customized communications to its clients. Moore BCS delivers personalized, easy-to-read documents that facilitate a positive impression on an organization’s customers. Its reengineering and redesign services help to ensure that the client organization’s business communications have high quality and clarity.

By outsourcing with Moore BCS, clients can divert their internal resources to other priorities because the dedicated
production facilities can be trusted to help ensure faster cycle times, ultimately reducing overall costs. Equipped with the latest print and digital technologies, Moore BCS has become a market leader in managing critical business communications.

Moore BCS offers products and services that include statement/billing, cards (e.g., phone cards, credit cards, etc.), government noticing, policyholder and plan member communication, and database marketing. The technology environment at Moore paces, and in many areas leads, the marketplace in its industry. Since its inception 30 years ago, Moore BCS has been regarded as one of the leading information processing outsource providers in North America.

Case Study Methodology

In the spring of 1999, we embarked on a case study of the card recovery system at Moore BCS. The goal of the research was to map the existing state of the card recovery system process. Once mapped, we were charged with redesigning the process to remove redundancies and improve the overall effectiveness of the system. We were not responsible for implementing suggested changes. Our job was to examine the overall process of the system and devise a set of recommendations. The study began in January 1999 and was completed in May 1999. We spoke with several BCS employees, but our main contacts were Ferris Jorgensen, Phone Card Project Manager; Dennis Elwood, National Manufacturing Systems Project Manager; and Harvey Black, Project Manager.

Our last meeting with Dennis was on December 4, 2000 to discuss future research and refine our theoretical assumptions. We keep in continuous contact when needed.

Analysis of the process was conducted in four distinct phases. Phase One was to define the problem. The problem was defined by articulating a problem statement. The problem statement was agreed upon by all parties involved and signed on February 24, 1999. The problem statement was as follows: “Moore Business Communication Services has a phone card division. During production, cards may become damaged or lost. The company has a need for a system that will track missing and replacement cards through the production cycle.” Keep in mind that a phone card recovery system existed prior to the study, but it was not fully automated.

During a meeting on March 3, 1999 with Ferris Jorgensen, it was decided that an updated system was needed to track missing and replacement cards through the production cycle. With the help of a small team of systems analysts and programmers, we were able to map the existing system, redesign it, and build a working prototype system that Moore can integrate into their existing information systems infrastructure.

Phase Two was to study the current physical system. This involved building entity relationship diagrams, data flow diagrams, statechart diagrams, and completing a feasibility study. Phase Three was defining end-user requirements. End-user requirements are the functional and technical needs of the logical new system. Phase Four was to clearly define the possible alternatives and select a feasible solution. After careful analysis and several meetings, it was decided to build a prototype using MS Access, MS Excel, and MS Visual Basic. The prototype has full functionality without building the necessary
error-checking routines. It can be integrated with existing BCS systems.

The March 3, 1999 meeting revealed the specifics of the existing process: “When a card is found missing or damaged, the operator fills out a missing slip form and turns it in to a central processor, who then enters the information into a spreadsheet and forwards the request for replacement cards to the programmers. A replacement card is then produced and inserted into the proper bundle to be shipped.” This process is not very well automated because it requires several people to communicate process changes continuously. As such, accuracy is suspect because of the tremendous potential for human error and efficiency is low because of the large amount of human observation needed to continuously monitor the process. The meeting also revealed the specific purpose of the redesigned system: “The purpose of the new application is to automate the card recovery process in an attempt to increase efficiency and accuracy. The new system should reduce the need for entering the original data several times. It should also make the entire process nearly “paperless” by eliminating several iterations of the same forms and information.”

The March 23, 1999 meeting revealed the system requirements: system inputs and functions, general system requirements, attributes to track, system outputs, and reports. System requirements are too numerous to mention, but we thought it prudent to include a few to give the reader a sense of the project’s scope. Some of the system inputs and functions included start number, end number, and enter missing number. Some of the general system requirements included implementing a virtually “paperless” process, eliminating forms, and building capability to determine where problems occur most frequently. Some of the attributes to track included programmer ID, project manager, client number, and workstation. System outputs included missing card “slip.” System reports included error occurrence and orders sent to programmers for replacement cards.

The design phase included acquisition and design of the newly mapped system. A request for proposal (RFP) was written to communicate to vendors the desired features and requirements. Its primary intent was to solicit specific configurations, prices, maintenance agreements, conditions regarding changes made by buyers, and servicing. It conveys proposals for evaluating criteria, closure, and postmark dates, and constraints. Meetings were held in early April to refine the RFP. The design specifications were agreed upon during April 1999. The design specifications explained the physical system requirements and the proposed prototype of the new system. The document included design of computer outputs, database and computer files, computer inputs, terminal dialogues and user interfaces, and methods and procedures.

The implementation phase included construction of the new system (prototype) and delivery of the new system. Meetings during April and May 1999 were conducted to facilitate this phase of the project. Construction included building, testing, and recording data and databases for the network of connected computers. It also included installation and testing of new software packages, and writing and testing new programs. Delivery included developing conversion plans including database installations, end-user training, and physical conversion. It also included the writing and delivery of the User Manual.
To provide a sense of how the process actually works, we now provide a brief description of the basic steps involved in the Moore BCS workflow as depicted in Figure 10. Manufacturing (Manufacturing Card) receives an order for cards. A decision is then made to check whether the cards are in raw form or already laminated. If the cards are not laminated, an image is copied; the cards are cut and then laminated. At this point, the cards are either sent for gluing (the image must be glued to a rigid backing) or they are sent to be bundled in a larger package and then glued. Once bundled, cards can be sent directly to Packing-Out for final delivery if the order requests this action. The same is true at Gluing Card. Once glued, the cards can be sent directly to Packing-Out. However, this is not the norm. Cards sent directly to Packing-Out are never scanned and therefore are not tracked properly in the system. Hence, it is recommended in most cases that all bundled and glued cards be sent to Scan/Bundle Card prior to Packing-Out. A check is then made at the end of Manufacturing Card (if the normal process flow is followed) for misplaced cards. As such, any cards suspected of being misplaced must be replaced. Data on
misplacement is sent to Recovering Missing Card so that the missing data can be retrieved and then properly replaced. From this point, all glued cards (if the normal process flow is followed) are sent to Scan/Bundle Card. Barcodes for cards are then scanned for proper storage in the database. Cards can be in either a single package or a bundle depending on the order request. Packages or bundles are then labeled and scanned. Scanning is done twice because bundles and packages have distinct barcodes from an individual card. Finally, bundles and packages are sent to Package-Out for shipment to customers.

Statechart Analysis

This article focuses on the statechart generated from the analysis of the card recovery process. We intend to analyze the statechart in terms of the three properties we defined earlier. With the aid of algorithms, we intend to examine the workflow in the context of statecharts to determine potential problems with the workflow. From our analysis and algorithmic generation, we begin building applied theory to assist systems designers in their attempts to design effective and accurate workflows.

In addition to the default conditions, additional conditions are shown in Figure 10. According to Harvey Black, those 14 extra conditions in Figure 10 (C1 – C14) are given by the user. However, these conditions must satisfy the following rules:

1. \( (C1 \oplus C2) = True \),
2. \( (C3 \oplus C4) = True \),
3. \( (C5 \oplus C6) = True \),
4. \( (C7 \oplus C8 \oplus C9) = True \),
5. \( (C10 \oplus C11 \oplus C12) = True \),
6. \( (C13 \oplus C14) = True \).

These rules specify that for the conditions in each rule, one and only one can be true at any given time. For example, in Rule 4, at any moment in time, there are only three possibilities, namely either \( C7 = True, C8 = False, \) and \( C9 = False \) or \( C7 = False, C8 = True, \) and \( C9 = False \) or \( C7 = False, C8 = False, \) and \( C9 = True \).

We now turn to some potential problematic structures in the statechart. Note that there are two cycles. One is from the high-level state Manufacturing Card to the high-level state Recovering Missing Card and back to Manufacturing Card. The other one is from the basic state Cello Wrapping Bundle to the basic state Gluing Card and back to Cello Wrapping Bundle. However, the event on the transition from Manufacturing Card to Recovering Missing Card is external. Therefore, the transition from Manufacturing Card to Recovering Missing Card will only take place when the event Missing Card information happens. We do not believe this event will happen all the time and thus the first cycle does not cause any critical problem. For the second one, since the transitions do not depend on external events and it is possible that \( C7 \) and \( C12 \) are both true, it is possible that non-termination may occur. We therefore advise the user that either \( C7 \) or \( C12 \) must eventually become false in the specification. Otherwise, there is a fundamental flaw in the workflow, that is, the cycle may never terminate.

Algorithm 1 provides a means to detect and correct non-terminating cycles. However, it provides much more. It forces us to analyze the statechart in a systematic
manner. Our first analytical action after we finished design of the Moore BCS workflow statechart was to identify cycles in the workflow. Once all of the cycles were identified, we then began to look closely at each cycle for possible non-termination. Without Algorithm 1, we would never detect or even suspect non-termination problems. Thus, Algorithm 1 acts as a high level analytical tool to systematically identify and correct non-termination problems in a given statechart. System designers can identify, discuss, and correct potential cycle non-termination problems during design rather than attempting to correct problems in production. Of course, system designers can also correct non-termination problems for existing systems in the manner discussed in the Moore BCS case.

Another potential workflow problem is confluence. Algorithm 2 provides a means to identify a set of transitions that may lead to non-confluence. In Figure 10, we identified an and-state in the sub-state Labeling contained in the high-level state Scan/Bundle card. According to Harvey Black, sometimes package labels are put on bundles or bundle labels are put on packages. Thus, the and-state Labeling may have a write-write racing problem.

Notice that we spoke with Mr. Black once we noticed the and-state. After Mr. Black was made aware of the and-state, he was able to better understand where the workflow problems were occurring. Currently, Moore is attempting to rethink the labeling process.

Hence, Algorithm 2 provided a systematic basis for redesign. By using the principles developed in Algorithm 2, we were able to flag the and-state structure as a source of potential problem and inform the user about it. Another potential workflow problem is observable determinism. As Theorem 5 indicates, if a statechart is not confluent and its outputs are visible, then it is not observably deterministic. Since the and-state Labeling has a write-write racing problem and its outputs are visible to the environment, the outputs of the and-state Labeling are not observably deterministic.

After explaining to Mr. Black the concept of observable determinism, he was able to identify a potential workflow problem. During label printing, there is a real danger that package labels and bundle labels can be switched. Although it is easy to distinguish by eye the difference between a bundle (a set of packages) and a package, it is very possible that an operator will accidentally place a bundle label on a package and vice-versa. Keep in mind that the labels are both plain white and the bar codes are not easy to see with the naked eye. As a result of this analysis, Mr. Black has suggested to management that bundle and package labels be made different colors. Although color printing is more expensive, the reduction in errors should more than justify the investment.

Algorithms 1 and 2 offer a systematic means to identify problems in complex business workflows. By using the principles developed in this study, one can scan any statechart quickly and efficiently to flag potential workflow problems. Process improvement and redesign has tended to focus on correcting, streamlining, and/or completely rethinking existing business workflows to reap vast improvements in performance and significant costs savings. However, this study pioneers the use of statechart analysis to identify workflow problems during redesign.

**FUTURE WORK**

We are in the process of developing
a workflow design tool that incorporates the ideas we developed in this paper with the notion of automating statechart analysis. The idea of developing this tool is to allow the user to develop sophisticated workflow models without having to be concerned with the underlying formalisms and algorithms we developed. In theory, the tool will automatically flag potential workflow problems for the user to aid in workflow redesign efforts. Of course we are speculating about the potential of our design tool until we can empirically validate it in the field. As such, we intend to conduct an extensive case study of Moore BCS once our workflow design tool has been prototyped. It is hoped that additional case study iteration will reveal the tool’s capabilities in a more granular manner.

We intend to further explore business workflows at Moore BCS and other organizations to validate and extend our findings. We recently visited (May 2001) an executive at Moore BCS who was not part of this study. The purpose of the visit was to initially verify the findings that we obtained from this study and discuss future work possibilities with Moore. The executive we interviewed was very positive about our current findings and has agreed to participate in an extension of this study.

Our next study will focus on developing the prototype and further testing our theory on a new workflow mutually agreed upon by us and the Moore BCS contact. If we can replicate the findings we obtained in this study, it will greatly enrich context and theoretical validity. As such, we hope to build a cumulative tradition over time.

Of course we realize that in-depth case studies tend to uncover many ideas, constructs, and concepts that are unanticipated. Therefore, we will try to keep our study somewhat within the scope of theory we have already generated to enable rigorous replication.

ENDNOTES
1 http://www.omg.org
2 See page 298 in Harel and Naamad, 1996.
3 http://www.aiim.org/wfmc/mainframe.htm

REFERENCES


Dr. Wai Yin Mok is an assistant professor of Management Information Systems at University of Alabama in Huntsville. His papers appear in *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, *Data & Engineering*, and *Information Processing Letters*. He is currently on the editorial board of *Journal of Database Management*. He reviewed papers for *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Software Engineering*, *Data & Knowledge Engineering*, and *The Computer Journal*. His research interests include XML scheme design, and business workflow systems.

Dr. David Paper is an associate professor at Utah State University in the Business Information Systems department. He has several refereed publications appearing in journals such as *Journal of Information Technology Cases and Applications*, *Communications of the AIS*, *Long Range Planning*, *Creativity and Innovation*, *Accounting Management and Information Technologies*, *Journal of Managerial Issues*, *Business Process Management Journal*, *Journal of Computer Information Systems*, and *Information Strategy: The Executive’s Journal*. He has worked for Texas Instruments, DLS, Inc., and the Phoenix Small Business Administration. He has performed IS consulting work for the Utah Department of Transportation and is currently consulting with the Space Dynamics Laboratory in Logan, UT. His teaching and research interests include process management, database management, e-commerce, business process reengineering, and organizational transformation.