CASH POINT WITH PROMELA/SPIN

BY WEI ZHOU, JEFFERY TANG
FEBRUARY 15, 2000

ABSTRACT

In this project, we develop a software controller for a cash machine system that involves multiple cash tills and users. The system is specified using Promela/SPIN, and it schedules and services requests to a central database, which may or may not be available. In this report, we describe the system requirements and the design decisions made in our model. Subsequently, we describe the distributed and concurrent implementation of the model. Furthermore, we mention some LTL properties that we prove with SPIN as well as the ones that we are unable to prove. Finally, we describe our experiences in the project and draw conclusion.

1. INTRODUCTION

This paper involves the discussion of a cash machine system using SPIN, which is a widely used model checker for supporting the verification of distributed systems. We choose to use SPIN because our system is distributed. In addition, we want SPIN to suppress details that are unrelated to process interactions [2]. The design, implementation, and verification of relevant LTL (Linear-time Temporal Logic) properties associated with the system are explained.

We take the informal definition of the cash machine system from a call for competition at the FM99 Exhibition [1]. The problem involves multiple cash machines and users in a system that schedules and services requests using a central database, which may or may not be available. The system is to provide concurrent access to the database from two or more tills, and to ensure that a transaction runs to completion once initiated by the user. Other requirements for the system include security to minimize the possibility of the use of stolen cards to gain access to an account, and reliability to guard against various hazards such as poor transmission between a till and the central database.

This document is organized into the following sections:

- Section 1. Introduction (the current section)
  Discusses the scope and overall organization of this document.
- Section 2. Cash Machine Specifications
  Provides background information regarding the project.
- Section 3. Modeling Design
  Talks about the major design assumption and decisions associated with the system.
- Section 4. Detailed Implementation
  Presents the detailed implementation of the cash machine system in SPIN.
- Section 5. LTL Properties
  Explains the verification relevant LTL properties of the system.
- Section 6. Conclusion

Describes our experience from using PROMELA/SPIN in this project and draws conclusion.

2. CASH MACHINE SPECIFICATIONS

The original specification of this project is included in the appendix of this report. Since the system description is not given in a formal language, there are ambiguities and omissions. Thus, we make several design decisions to the specification to make it concrete. As seen in Figure 1, the central database is connected to many cash machines, and one of the most important design goals is to allow concurrent access to the database. The cash machine services one customer at a time by taking in a bank card.

We assume that the cash machine system has TILL_NUM number of tills and USER_NUM number of cards. Each card stores the information of user ID, password, daily withdraw limit and an account number associated with this card. The central database keeps the account balance for each account in this system identified by account number. After the system initiation, each card is set to have a balance of INIT_BALANCE and the daily withdraw limit of DAILY_LIMIT. The system resets the daily withdraw limit to simulate the opening of another transaction day. At any given time, the availability of the central database is non-deterministic.

Transactions on a card are only allowed after the card passes authorization at a till. The user provides both user ID and password to a till, and the till encrypts the password and compares it with the code stored in the card. After continuous ILLEGAL_TRY times of unsuccessful logging attempts, the card will be kept by the till. We allow cardholders that are authorized to use the same account to login through different tills at the same time. Thus, an account can be accessed by more than one card simultaneously.

After authentication, the cardholder is provided with the options to ask for a statement, check account balance if the central database is online, or withdraw cash. The amount of each withdrawal is set to a fixed number - WITHDRAW_AMOUNT. In the case when the database is online, a withdrawal is allowed if WITHDRAW_AMOUNT is less or equal to the daily limit on the card as well as the associated account balance stored in the database. In the case when the database is offline, a withdrawal is allowed if WITHDRAW_AMOUNT is less or equal to the

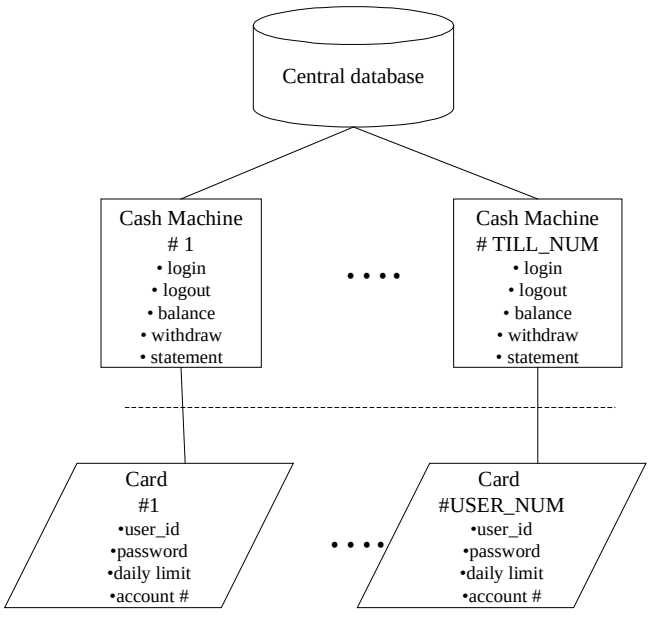

Figure 1: Cash Machine System

daily limit on the card and the withdrawal amount accumulated during the period of database offline is less than UPPER_BOUND. When the database becomes online, the accumulated amount of money will be subtracted from the balance of the account. The minimal balances for any account is zero. When the cards are reset at the beginning of a new transaction day, if the balance of the account is zero, the system resets the balance to INIT_BALANCE. This simulates a deposit and allows the card to make further withdrawals when the central database is online.

3. DESIGN

Based on our previous experiences with SPIN, we put a lot of emphasis on designing a system for better performance and avoiding the problem of state explosions. We come up with two models: one with a centralized controller serving all the tills; and the other with multiple controllers, one controller per till.

In this section, we will first briefly present the design of the centralized-control model in "An Alternative Design"; then we will elaborate the design and the implementation of the multi-controller model in details.

AN ALTERNATIVE DESIGN

We start from rather a simple system architecture with centralized control. In this architecture, as shown in figure 2 below, we use multi-instances of a proctype - *till* to simulate user input using a bankcard from different cash machines. An instance of the proctype *control* simulates the processing of transaction requests coming from different tills. Each *till* has its own queue *queue_reply* to receive results of transactions from *control*. All the requests from different tills are queued in a global queue *queue_request*. The *control* has access to the central database. The maximal length of *queue_request* is set to TILL_NUM, since a till has to wait for a response of the previous request before sending out another one. This design circumvents the situation of mutual exclusion when an account is accessed by more than one card at the same time. It, in a sense, uses a time stamp to differentiate requests. On the other hand, this simplicity sacrifices performance, and main *control* becomes the bottleneck and reduces the throughput of the system. In comparison to a distributed control system, the performance is lowered by the factor of TILL_NUM times.

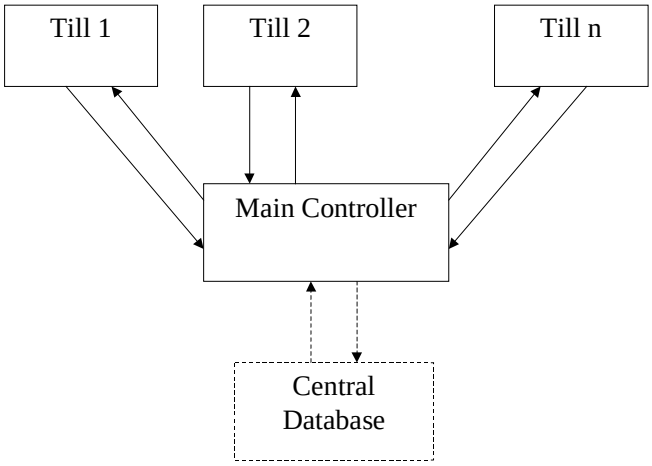

Figure 2: Alternative Design Using a Centralized Controller

MODELING

After implementing the first design shown in figure 2, we evolve the system using a distributed approach (see Figure 3 below) by adding a mutual exclusion lock into *control* and instantiating multiple instances of *control* at the system initialization. There are two ways to implement the mutual exclusion.

1) Use PROMELA to provide the built in *atomic* operation [3].

   atomic is used to define a block of code where interleaving by other proctype is not allowed. In this implementation, we use *atomic* to bound each writing operation to the daily withdraw limit and the account balance.

2) Use a read/write flag on each card and each account.

   In this alternative approach, we define an extra field as a mutex for each card. Before the actual writing operation happens, a proctype has to wait until the flag is in READ state and set the flag to WRITE after entering the critical section.

It is obvious that the latter is more efficient for the system performance than the former one, since in the former method all the other proctypes are blocked from entering the critical areas defined by *atomic* after one proctype enters first. However, the latter one is more complicated both in terms of implementation and verification. It increases the number of states exponentially. We stay with option 1 for the design of this system.

The components in the cash machine system are shown in Figure 3. Note that the boxes with solid lines indicate processes, while the boxes with dashed lines indicate entities. Communication between components is illustrated using arrows.

Discussion on the design rationale for user entity(*till*), controller process, database entity, and day-timer process follow.

The design of the model is heavily influenced by the need to minimize the state space requirement for verification. Since SPIN needs to analyze the full state space of the system to determine the validity of LTL properties, it is essential that the least number of bytes encode the system states for both the memory and the verification time purposes.

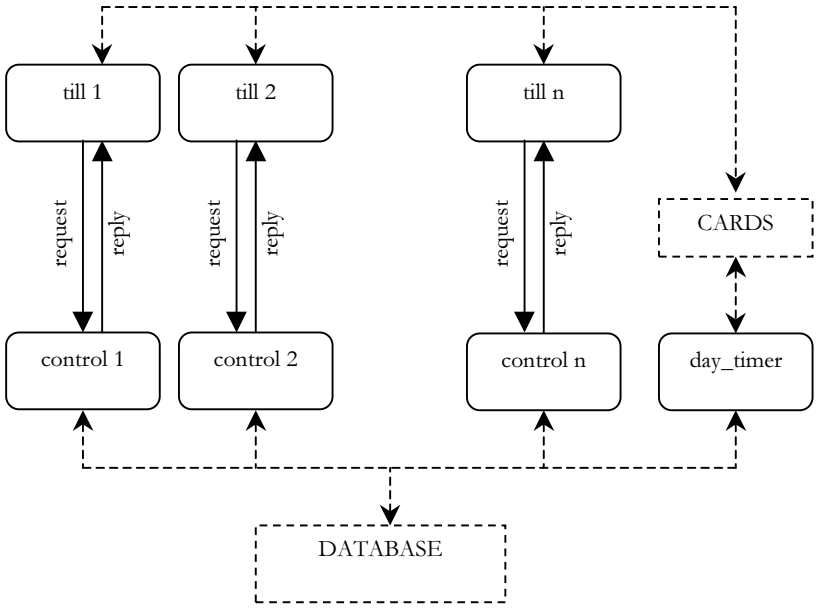

Figure 3. Components of the PROMELA Model

CARD

As mentioned in the design considerations, each card stores information on user id, password, daily withdrawal limit as well as a temporal variable which keeps track of the total withdrawal amount when the database is offline. In order to save the memory usage as well as to reduce the number of state, we use *byte* type for all the variables and carefully prevent the data from overflowing in the implementation.

ACCOUNT

We store the account balance in a *byte* variable. Each account has an *account_id* to identify itself.

TILL

We use *till* to model the valid user input with a card. By *valid* user inputs, we mean that the user input should follow a certain logic, for example, user cannot withdraw or perform any other transaction unless the card has been authenticated during login by the *control* first. The possible commands of user inputs are *Login, Logoff, Statement, Balance, Withdraw*. See the details of how proctype *Till* and proctype *Control* handle these five kind of requests in the "Implementation" below.

CONTROL

We use *Control* to process the user requests coming from *Till*. It serves as a bridge from the user interface *till* to the central database. We use several *atomic* blocks to inhibit interleaving within critical sections. Since we keep certain logic within *Till*, *Control* in principle does not check the validity of the type of a request coming in; rather it validates the parameters of the request.

DAY_TIMER

Day_timer is used to randomly change the status of the central database: online/offline. It also simulates the opening of a new transaction day, resets the daily withdraw limit for each card, and sometimes resets the account balance as described in the design considerations.

IMPLEMENTATION

This section describes the implementation of our PROMELA model. Three main aspects of our implementation are discussed. They are the data structure used to encode the system states, the message format used for process communication, and the processes themselves.

DATA STRUCTURE

The data structure is defined with considerations for minimizing the number of system states. There are two main data types in the system, Till and Card.

Till

Tills are represented by the TILL data structure containing the following components:

ID	A byte that uniquely identifies the till
REQUEST	A channel that is used to send request to the controller for processing
REPLY	A channel that receives messages from the controller to get the result of the previous request

The three attributes above are used to build the functionality of this application. The rests are used only for verification purposes.

TILL_REQ	A byte that stores which request is sent out to *Control*.
TILL_RES	A byte that stores which response is received out to *Control*.
STATUS	A bit that shows whether the card has passed the authentication or not.
User_id	A byte that stores the user_id after authentication.
Msgdata	A byte that shows the parameter sent to *Control*.

Count A byte to indicate the number of continuous login failures in order to identify the illegal card.

All tills are in an array (m_till) and can be individually accessed through array indices.

Card

Users are represented by the USER data structure containing the following components. Note that since the user carries the cash card, the user structure also contains the card information.

UID A byte that uniquely identifies the user

UPWD A code stored on the cash card for identifying the card owner

BALANCE A number amount indicating the current balance in the account

DAILY_LIMIT A fixed number that limits the amount of money that could be withdrawn on a daily basis

CUR_USER A byte that is used to handle the case where concurrent attempts from two or more card holders who are authorized to use the same account

TMP A byte that is used to store the amount that has been withdrawn from the corresponding account when the database was offline

A global constant that indicates the number of users in the system is defined, and all users are stored in an array and are accessed through array indices (i.e. USER m_user[USER_NUM])

DB flag

This is a boolean flag to show whether the central database is online or not.

PROCTYPES

There are a number of interacting processes in the system. They are the Till Process, the Controller process, and the Day-timer Process. This section discusses each of the processes in detail.

TILL

The Till Process is responsible for getting requests from the user and perform actions based on the request and the current status of the user. The flow chart of the Till Process is shown in Figure 4.

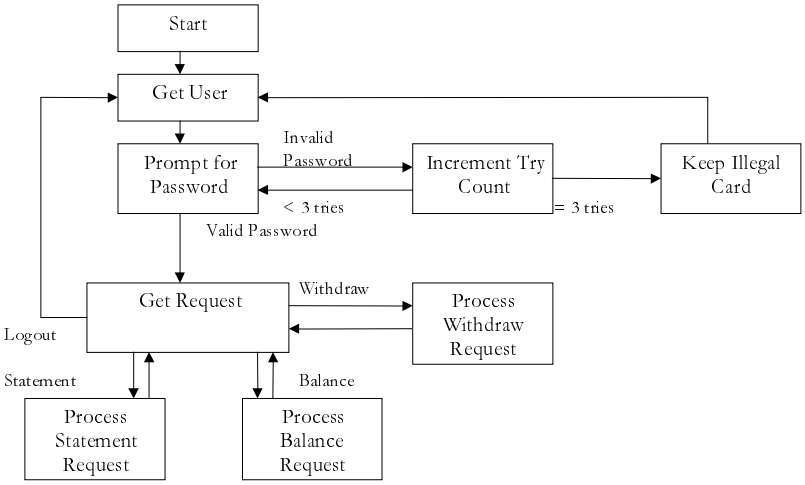

Figure 4: Flow Chart for the Till Proctype

As seen in the above figure, the Till Process starts with accepting a user's *login* request and prompting for a password. If the password entered does not match the number encoded on the card, the till increments the invalid try count. This is done for security reasons; allow only the rightful owner of the card to access the bank account. If the PIN is incorrect entered for the card three times in a row, the till issues an ILLEGAL_CARD message and keeps the card. To verify the validity of the password, the password is encrypted by the till and compared to the code stored on the card. We use a primitive encryption method that simply increases the user inputted password by the ASCII value of 'a'.

If the correct PIN is entered for the card, the till provides four options the user: *withdraw, balance, statement, or logoff*. The user can perform more than one transaction when in the system, and he/she signs off by explicitly requesting logging off. The till will then eject the card and start serving the next user. When a user is in the system, the requests are not processed by the till locally, but are sent to a controller that has access to the central database. The result of a transaction is passed to proctype *Till* through the *Reply* channel(see the definition of data structure above). There are *OK and FAIL* results for each type of transactions respectively. See "Message" below for details.

Control

As discussed in the design section, each *till* is connected to a controller to allow maximal availability. The flow chart for the control process is show in Figure 5. The manipulation of the *withdraw* transactions is of particular interest. When the database is online, the system verifies if the amount entered is less than the daily limit and the current balance before performing the withdrawal. When the database is offline, the system is unable to query the database for the account balance. In this case, the system stores the amount for withdrawal in a temporary variable and deducts the amount when the database comes back online. Thus the system is robust enough to guard against hazards such as a poor transmission between a till and the central database.

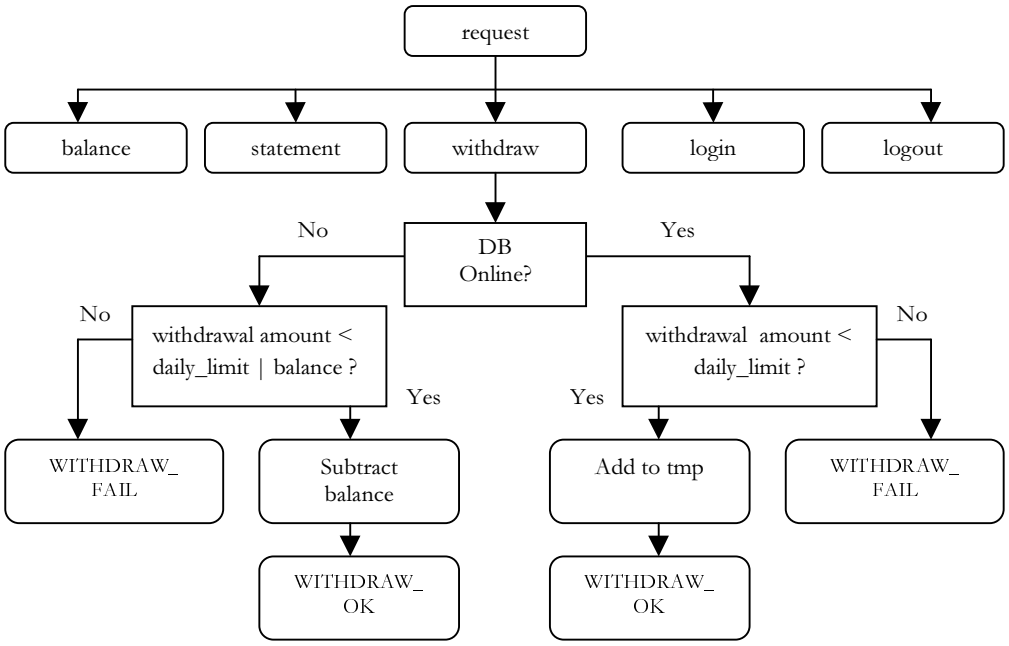

Figure 5. Flow Chart for Control Process

DAY_TIMER

This proctype carries out the two tasks of this system. The first task is to simulate the availability of the database to the tills. The database is not available if it is offline, or if the connection between the tills and the central database is interrupted. Thus, the DAY_TIMER process non-deterministically sets the database availability flag to online or offline. The second task is for resetting the daily withdrawing limit for each account to indicate the beginning of a new day. We combine these two unrelevant tasks together in one process for simplicity. In addition, separating them into two processes would use extra computation resources in terms of task scheduling and executing time. The combination would not affect the properties from the application point of view.

MESSAGES

It is important to have a clear definition of the type of messages that are passed back and forth between processes. For example, we need to keep track of the current status of each user to know what kinds of operations are permitted at each stage. These messages provide a means of coordination between processes.

The messages can be divided into two main categories: the request message types and the reply message types. The request messages are partitioned based on the action to be formed, and the reply messages are used to indicate whether the request is successful or not. By associating the request messages with the reply messages, we are able to confirm the completion of transactions.

Request messages are actions that the initiating process would like to perform. The list of request message types is shown below:

LOGIN	To login into the cash machine system
BALANCE	The request to view the bank balance online
WITHDRAW	The request to withdraw cash from the customer's account
STATE	The request to get the statement to be sent by mail

Response messages are used to return the status of the previous requests. The list of response messages is shown below:

LOGIN_OK	The request to login is performed successfully
LOGIN_FAIL	The request to login is performed unsuccessfully
BALANCE_OK	The request to view the bank balance online is performed successfully
BALANCE_FAIL	The request to view the bank balance online is performed unsuccessfully
WITHDRAW_OK	The request to withdraw cash is performed successfully
WITHDRAW_FAIL	The request to withdraw cash is performed unsuccessfully
STATE_OK	The request to have statement sent by mail is performed successfully
STATE_FAIL	The request to have statement sent by mail is performed unsuccessfully

4. PROPERTIES

The system specification describes two kinds of properties: one is time-related properties, the other is behavior-related properties. A typical time-related property is "all the requests should be processed within a given time." Unfortunately, LTL, the temporal logic used in SPIN, does not provide a mechanism for defining a time span and hence cannot define a performance-related property.

There are two categories of properties that can be verified in temporal logic:

1. Safety – nothing bad ever happens
2. Liveness – something good will eventually happen

We verify the following properties for the cash machine system. In order to verify the properties, we need to define several extra global variables to label the state of the model and make some of the local variables globally visible.

The result of the verification is shown in table 1 below.

For safety

Prop 1. A card is labeled as illegal card after ILLEGAL_TRY times of failing attempts of login in a row.

The following property verifies that for a given till, in this case till #0, it is possible that the internal counter for counting login failures is greater or equal to the ILLEGAL_TRY. And it is always true that in such case, the till will set the response to the login request to ILLEGAL_CARD.

$$\Diamond(m_tills[0].count \geq ILLEGAL_TRY) \;\&\&$$
$$\Box \;\; (((m_tills[0].count \geq ILLEGAL_TRY) \;\&\&\; (m_tills[0].till_req == LOGIN)) \rightarrow$$
$$\Diamond(m_tills[0].till_res == ILLEGAL_CARD))$$

Prop 2. The account balance is always between 0 and INIT_BALANCE.

The following property verifies that for a given card that is matched to a user, in this case the card for user #0, it is always the case that the balance of this card is always between 0 and INIT_BALANCE, the maximal balance of the card.

$$\Box((m_users[0].balance \geq 0) \;\&\&\; (m_users[0].balance \leq INIT_BALANCE)$$

Prop 3. Request "View Balance" is invalid or is rejected when the central database is offline.

The following property verifies that it is possible that the central database could be offline. And it is always true that in such case, for a given till, in this case till #0, the *balance* request sending from this till would be rejected by the *Controller* and the *Till* would receive a BALANCE_FAIL response.

$$\Diamond(m_dbavailable == FALSE) \;\&\&$$
$$\Box\ ((m_dbavailable == FALSE) \;\&\&\; (m_tills[0].till_req == BALANCE)) \rightarrow$$
$$\Diamond\ (m_tills[0].till_res == BALANCE_FAIL$$

Prop 4. Withdraw attempt will always fail when withdraw amount is greater than daily limit.

The following property verifies that it is always true that when the current request of a given till, in this case till #0 is *withdraw* and the till is serving the user with user id of *user_id*, and if the daily limit of the user is less then the withdraw amount stored in *msgdata*, the *withdraw* request would be rejected by the *Controller* and the *Till* would receive a WITHDRAW_FAIL response.

```
□ ((m_tills[0].till_req == WITHDRAW) &&
(m_users[m_tills[0].user_id].daily_limit < m_tills[0].msgdata)) →
◊(m_tills[0].till_res == WITHDRAW_FAIL)
```

Prop 5. Withdraw attempt will always fail when database is online and withdraw amount is greater than the account balance.

The following property verifies that in case the central database is online *m_dbavailable = TRUE*, it is always true that when the current request of a given till, in this case till #0 is *withdraw,* the till is serving the user with user id of *user_id*, and if the *balance* of the user is less then the withdraw amount stored in *msgdata,* the *withdraw* request would be rejected by the *Controller* and the *Till* would receive a WITHDRAW_FAIL response.

```
□ ((m_tills[0].till_req == WITHDRAW) &&
(m_users[m_tills[0].user_id].balance < m_tills[0].msgdata) &&
(m_dbavailable == TRUE)) →
◊(m_tills[0].till_res == WITHDRAW_FAIL)
```

Prop 6. The request from a till can only be LOGIN, BALANCE, STATE, WITHDRAW.

The following property verifies that it is always true that a given till, in this case till #0 the request sends to the *control* cound only be LOGIN, BALANCE, STATE, WITHDRAW.

```
□ ((m_tills[0].till_req == LOGIN) || (m_tills[0].till_req == WITHDRAW) ||
(m_tills[0].till_req == BALANCE) || (m_tills[0].till_req == STATE))
```

Prop 7. In a till, before a request is received and responded, no other request should be sent to *Control.*

The following property verifies that it is always true that a given till, in this case till #0, when there is a request queuing in *request* queue of a till, the *response* queue of that till is empty.

```
□((len(m_tills[0].request) == 1) && (len(m_tills[0].response) == 0))
```

For Liveness

Prop 8. All requests will be processed eventually.

The following property verifies that it is always true that a given till, in this case till #0, when there is a request queuing in *request* queue of a till and the *response* queue of that till is empty, sometime in the future the request would be taken by the *Control* and a response would be put into the *reply* queue.

□ *(((len(m_tills[0].request) == 1) && (len(m_tills[0].reply) == 0)) →*
◊((len(m_tills[0].request) == 0) && (len(m_tills[0].reply) == 1)))

The verification results turn out to be very good. To save memory, we use *DBITSTATE* when compiling. We set 640MB as the maximal memory available (most of the properties we verified took about 400MB.), and 10^7 for searching depth. Below is a table for the verification results.

Property #	1	2	3	4	5	6	7	8
Time(min)	27.36	27	27.31	25.23	26.36	27.57	27.46	29.03
Mem(MB)	488.074	488.074	488.074	161.95	488.074	486.538	485.538	491.074
Transitions	5.25E+07	5.25E+07	5.25E+07	5.79E+07	5.25E+07	5.33E+07	5.33E+08	5.49E+07
Depth	1936636	1936636	1936636	899999	1936636	1944494	1944494	1974536
States	3.19E+07	3.19E+07	3.19E+07	3.04E+07	3.19E+07	3.23E+07	3.24E+07	3.31E+07

Table 1: Results of Property Verification

5. CONCLUSION

In this project, we used PROMELA/SPIN to implement a cash machine system that involved multiple tills and users. We opted for a distributed approach that allowed concurrent user accesses for maximal availability in the model. During the implementation of the project, we realized that concurrent processes are hard to model correctly, even for a simple system. We kept on running into memory shortage when we tried to verify the LTL properties, and to get around the problem, we strove to limit the size of the model to reduce the number of state variables that encode the system. The positive side we found about SPIN is its expressiveness for checking properties.

We encountered two major issues when designing and verifying the system. The first one was the haunting problem in model checking – the state explosion problem; the second one was related to simulation.

[3] explains "There is no random number generation function in Promela. The reason is that during verification, SPIN effectively checks for all possible executions of a system. Having even a single occurrence of a call on the random number generator would increase the number of cases to inspect by the range of the generator. Random number generators can be useful on a simulation, but they can be disastrous when allowed in verification." For this reason, we limit the selection of users' inputs on user_id, password, and withdrawal amount. We initially configured the system to have four users, five password selections, and three options for the withdrawal amount. We also simulated the deposit features to make this banking service complete. During verification, with a memory space as much as 1G, SPIN still reported an"out of memory" error. To work around, we removed the Deposit option, which was not required by the specification and which was oddly considerable to the state explosion issue: because *Deposit* was not a stand-alone feature, it interwove with *Withdraw* and furthered the multi-writer problem in the model. We curtailed the configuration of our model to the minimal multi-user configuration, reducing

USER_NUM to two and TILL_NUM to two. We also set the withdraw amount to a fixed number. By doing this, the total number of states was cut at least in 2*2*3 = 12. Eventually, we are able to use around 400M of memory to verify the properties.

We encountered another difficulty during the design and testing phase. We designed two seperate proctypes to simulate the changing of transaction days and to determine the availability of the central database. In self-testing, we found out that these two proctypes took a large protion of the computing resources; but other proctypes, such as *till* and *Control* with more complicated logic were always suspended for the lack of computation resources. We had to abandon the thought of adding more proctypes into the model, such as a proctype to simulate the users lining up in front of the tills, and a proctype to simulate a *statement* processing center. To solve this problem, we merged the proctype of day turning and that of the database availability into a single proctype, since that this merge would not effect the behaviors of the system. It also helps to reduce the number of transitions, because of less interleaving between proctypes.

REFERENCES

[1] FM99 Exhibition: Competition "Cash-Point Service".

[2] G.J. Holzmann, "The Model Checker SPIN", IEEE Transactions on Software Engineering, 279-295, Vol. 23, No. 5, May 1997.

[3] SPIN On-line Reference http://netlib.bell-labs.com/netlib/spin/whatispin.html

[4] W. Thomas, "Automata on Infinite Objects", Handbook on Theoretical Computer Science, J. Van Leeuwen, ed., pp. 133-187, Elsevier Science, 1990.

[5] M. Vardi, P. Wolper, "An Automata-Theoretic Approach to Automatic Program Verification", in Proceedings of First IEEE Symposium on Logic in Computer Science", pp. 322-331, 1986.

[6] M. Vardi, P. Wolper, "Reasoning about Infinite Computations", in Information and Computation, vol. 115, pp. 1-37, 1994.

APPENDIX A. ORIGINAL SPECIFICATION

FM99 Exhibition: Competition

<< Cash-point>> Service

Background

The task which is the basis for this competition has been taken from a workshop on <<The Analysis of Concurrent Systems>> held in Cambridge in September 1983. The description of the task has been updated slightly to reflect advances in technology and understanding.

The 1983 workshop was reported in LNCS 207. The organizers formulated ten <<problems>> in formal specification and invited some twenty six leading computer scientists to present solutions. There were thirty three solutions in total, but none were provided for the <<ninth

problem>>, although it did not present any special difficulty compared with others. This competition for the FM99 exhibitors is based on that ninth problem.

The Task

You are invited to specify the service described below using your formalism, and to user your tool to process that specification. Proofs of properties related to security of transactions would be especially interesting. You should explain what benefits result from using facilities of your tool. Since the following description is not in a formal language, there will be ambiguities and omissions. You should resolve these, not by contacting the organizers, but by internal debate within your own organization. In your presentation of the results you should include a brief account of these questions and their resolutions.

You should prepare an informal 20-minute presentation of your solution suitable for presenting at your exhibition stand. A written-up version of your solution should, if possible, be included in hand-out material.

Description

There are many tills which can access a central resource containing the detailed records of customers' bank accounts. A till is used by inserting a card and typing in a PIN (Personal Identification Number) which is encoded by the till and compared with a code stored on the card.

After successfully identifying themselves to the system, customers may try to:

1. View the balance of their accounts

2. Make a withdrawal of cash

3. Ask for a statement of their account to be sent by post

Information on accounts is held in a central database and may be unavailable. In that case 1) above may not be possible. If the database is available, any amount up to the total in the account may be withdrawn, subject to a fixed daily limit on withdrawals. This means that the amount withdrawn within the day must be stored on the card.

<<Illegal>> cards are kept by the till.

The solution should consider feasible implementation architectures. These are likely to include a data line between each till and the central database. For example, transactions at a till may be recorded locally and sent in a queue down the line to the central database.

Concurrency, time constraints

You should be able to show that the solution allows concurrent access to the database from two or more different tills. It is important that once a user has initiated a transaction, the transaction is completed at least eventually, and preferably within some real time constraint.

You should also consider how to handle concurrent attempts from two card holders who are authorized to use the same account.

Security, reliability

Clearly, for this sort of application, it is important to minimize the possibility of the use of stolen cards to gain access to an account, and that implementations of the system are robust against various hazards such as poor transmission between a till and the central database. Any way the solution can address such issues will be particularly welcome.

Proofs

Any proofs or demonstrations (by, e.g., animation or model checking) of concurrent accessibility, temporal properties and security and reliability aspects outlined above will be especially welcome, as will proofs of any provided reification.

Abstraction

Notwithstanding the above, the specification you provide should be reasonably abstract. A feasible design for an implementation may be presented at a first stage reification of the abstract specification.

Prize

The solutions will be judged by a panel of three members of FME who will visit the exhibition stands of the competitors and hear their presentations. The prize for the winning entry will be a refund of the exhibition registration fee that the winning competitor has actually paid, with a maximum of 2500 Euro. The entries will be judged according to how well they meet the objectives outlined above. The judging panel will be independent of any competitor organizations.

Exhibitors wishing to take part in the competition should inform Nico Plat by Wednesday 15th September. A schedule of the competitors' 20 minute presentations will then be published on the conference notice board on Monday 20th September. This schedule will enable the judging panel to visit each presentation.

The winning solution will be announced in a special session on the last day of the conference. The organisers cannot enter into discussions about the selection of the winner.