A note on recoverability of modular systems*

by PHILIP M. MERLIN and DAVID J. FARBER

University of California
Irvine, California

INTRODUCTION

In a paper by Gostelow-Van Weert, it was shown that any processes can be described by Petri-nets (PN). Thus PN can be used as a model in the design (or analysis) of any computer system. This approach makes it possible to utilize all the theoretic knowledge developed for PN and thus provides a powerful tool for computer systems (or program) design. In this case, the PN can be used as a framework that provides a tool for:

1. the understanding of the design,
2. handling the incremental changes of the design (without redesigning the entire system),
3. the implementation of desired properties (like recoverability in case of failures, proper termination, mutual exclusion, etc.), and
4. at least, the analysis of the program in order to check if it has the desired properties.

In Reference 2, it was shown that the UCLA Graph Model of Computation and the Petri-nets are equivalent. This means that the same approach can be applied using PN, the UCLA Graph Model, or any other equivalent model of computation. Note that these models are able to represent a concurrent environment (multiprocessing, multiprogramming, computer networks, etc.).

In the past, the use of these models as a practical tool was limited. The principal reason of this limitation is that any program (or system) of reasonable size has a corresponding very large model, usually of unacceptable size, even for processing by computer.

In order to simplify the handling of these models of programs, many researchers tried to find a way of dividing the models into modules. We note that the models of structured (modular) programs are easily divisible into natural modules, the subroutines. Then, any structured, hierarchical, top-bottom, modular design can be relatively easily modeled by Petri-nets, et al. At each level, the description of the desired system behavior is given. This description is then modeled by a Petri-net and analyzed using the theory developed for Petri-nets.

We note that distributed computation naturally leans toward structured (modular) programming and indeed makes it difficult to do otherwise. We feel that the application of the above modeling methodology will enable one to gain strong insight into this and other areas.

In the following sections, we will focus our discussion on the problem of recoverability. Our approach is based on the method for analyzing and designing recoverable Petri-nets presented in References 3, 4 and 5. The same method is improved in Reference 6, and an example of its use is given in Reference 7. By “recoverability” we mean that after the occurrence of a failure the control of the process is not lost, and after several steps it will return to “normal execution”. Note the difference between this concept and the concept of “correctness of results”. In this paper we do not deal with the problem of correctness. We are concerned with control recoverability—structural recoverability. Our approach is motivated by the philosophy that we can accept the situation in which a user gets some erroneous results, but we do not accept the possibility that a single error (or failure) may cause the entire breakdown of big systems.

In the next section a short background is presented. A later section presents an example, the top level design of a simple operating system.

RECOVERABILITY OF A PETRI-NET

Petri nets model “conditions” represented by nodes and “events” represented by transition bars. The holding of a condition is represented by placing a token on that node. Directed arcs connect nodes to bars and bars to nodes. A transition bar (event) can fire (occur) if all the nodes (conditions) input to that transition bar have tokens (hold). When a transition bar fires, it removes one token from each input node and places one token on each output node.

If we permit repetitions of the directed arcs, the number of tokens placed (or removed) in each node is equal to the number of repetitions of the arc connecting the transition bar and the node.

Figure 1 shows a PN example. Assuming as initial condition a token in node 1, bar 1 can fire. When bar 1 fires it removes the token of 1 and places a token on 2 and two tokens on 3. In this new state (2, 3, 3) only bar 2 can fire. If bar 2 fires it removes a token from 3 and places a token on 4. In this position bar 2 or bar 4 can fire, and so on.

The state of the Petri-net is defined by the collection of names of the nodes holding tokens. The number of
instances of a node in a state is equal to the number of tokens the node holds in this state. All the possible states in which a PN can stay and the possible transitions between them define a state machine called Token Machine (TM). The TM for the net of Figure 1 is shown in Figure 2, assuming as initial state a token in 1.

Suppose that a condition in the PN may fail. In this case, a token held by this condition may disappear. It is possible to represent this characteristic by adding a new branch to the TM. This branch will represent the possible flow of the execution when the “problematic” token disappears. For illustration, suppose that when the PN of Figure 1 is in state (2, 4, 5) a token of 5 disappears, then the PN will go to state (2, 4). In state (2, 4) bars 3 and 4 can fire. A fine state machine including the TM of figure 2 and all the additional paths for the case that 5 can lose a token is given in Figure 3. Thick lines represent the TM and thin lines describe the paths added since a failure. This new machine will be called Error Token Machine (ETM).

The states (and transitions) of the TM are called "legal states" ("legal transitions") and the other states (transitions) are called "illegal states" ("illegal transitions"). At this point we can state the conditions for recovery:

"A process P is recoverable from failure F if and only if in the ETM of P for failure F, all the directed paths through illegal states arrive to legal states."

It means, after a failure, the execution sequence must return to normal execution after a finite number of steps.

In this work, we limit our study to processes that have finite TM. From the properties of directed graphs, for the case of finite TM we can derive an equivalent set of conditions for recoverability:

"a process P is recoverable from failure F if and only if in the ETM of P for failure F:
1. the number of illegal states is finite,
2. there are no final illegal states, and
3. there are no directed loops including only illegal states."

In this paper, these conditions are named Conditions of Recoverability (COR). Figure 3 shows that the PN of Figure 1 is not recoverable from failures in 5. Conditions COR are not satisfied because of the final illegal state (2, 6) and because of the loop between the states (2, 4) and (2, 5).

In the previous discussion we deal only with the case that a token may disappear. But in the same way, because of a failure, a condition in the PN may generate a token. This situation may also be represented by adding new branches to the correspondent nodes in the TM. The approach is similar to the previous case.

In References 4 and 6, it was shown a method of designing recoverable PNs for failure of kind "loss of token". The design is executed in two steps. First, a TM that can be implemented by a recoverable PN is designed. In the
second step, a recoverable PN corresponding to the given TM is generated. In order to accomplish this method, develop a way of designing PNs that affect a given TM, and also gives the necessary and sufficient conditions that a TM has to satisfy in order to be implementable by a recoverable PN. Since the TM of Figure 2 contradicts these conditions, this TM cannot be implemented by a recoverable PN. In References 5 and 6, it was shown that in case where no assumptions have been made about the execution times of the different parts of the PN, the recoverable processes under a failure of type “loss of token” are very limited in their possible structures. These limitations are usually unacceptable in practical processes. Because of these limitations, some knowledge about the execution times was introduced in the PNs, and a new model, the TPN, was defined. In this new model, practical processes can be implemented as recoverable, but in this recoverable TPNs it is necessary to state some constraints on the execution times of its parts. The reader interested in a more detailed and formal discussion on the TPN and recoverability of the TPN is referred to the sources. Here we limit ourself only to the discussion “how to design a recoverable TPN that implements the TM of Figure 2".

A TPN is defined by a PN where for each transition bar \( b_i \) a pair of real numbers \( (t^{*i}; t^{**i}) \) are given. In a TPN a bar \( b_i \) can fire only after its input conditions hold for a period of time larger than \( t^{*i} \). On the other hand, if the input conditions of bar \( b_i \) hold for a period of time \( t^{**i} \), then \( b_i \) must fire. Note that in some sense, \( t^{*i} \) and \( t^{**i} \) give a measure of minimal and maximal execution times of the events (the executions). Note that when no constraints are stated for these times (for all \( i; t^{*i}=0 \) and \( t^{**i}=\infty \)) the TPN is equal to a PN.

In order to transform the PN of Figure 1 into a TPN that satisfies the conditions COR and has the TM of Figure 2 two changes have to be introduced. First, the illegal loop of states (2, 4) and (2, 5) have to be eliminated. Second, (2, 6) has to be converted into a no final state. The loop of states (2, 4) and (2, 5) is broken if bar 4 is split into four different bars, as shown in Figure 4. Note that the TM of the PN of Figure 4 is the same TM of Figure 2, but its ETM (see Figure 5) differs from the ETM of Figure 3. References 4 and 6 present an analysis on the ways of breaking illegal loops.

In order to transform state (2, 6) into a non-final state it has to be a bar, say 7, that fires in state (2, 6) bringing the system to one of its legal states. For example, bar 7 can be defined as:

- input conditions of 7 = 2, 6
- output conditions of 7 = 1

This new TPN is shown in Figure 6. But, if the TPN of Figure 6 is supposed to have the same TM of Figure 2, then bar 7 has to be prevented from firing in any state out of (2, 6). In other words, \( t^{*7} \) has to be bigger than the maximal time that the conditions 2, 6 can be enabled in the TPN. If this time is denoted as \( T^{**26} \), then \( t^{*7} \) has to satisfy:

\[
t^{*7} > T^{**26}
\]

In this case, bar 7 can fire only after it is sure that the TPN is in state (2, 6).
It is possible to show that $T^{*26}$ satisfies:

$$T^{*26} \leq \max(t^{**}2 + \min(t^{**}3; t^{**}4) + t^{*6}; t^{**}5 + \min(t^{**}3; t^{**}4) + t^{*6})$$

Thus, if:

$$t^{*7} > \max(t^{**}2 + \min(t^{**}3; t^{**}4) + t^{*6}; t^{**}5 + \min(t^{**}3; t^{**}4) + t^{*6})$$

then:

$$t^{*7} > T^{*26}$$

is satisfied.

A general study on the problem of eliminating final illegal states is presented in References 4, 5, and 6.

The TPN of Figure 6 with the constraint (1) has the TM of Figure 2 and it is recoverable after the occurrence of a failure of kind “loss of token” in condition 5.

The next section demonstrates a practical example of the use of the methodology presented.

EXAMPLE—THE DESIGN OF A SIMPLIFIED O.S.

In order to present an example big enough to show our methodology, but small enough to fit into the physical limitation on papers for this conference, we decided to present a simplified version of an operating system (OS) design. We admit that our example is much simpler than any “real” OS, but it shows the way of designing we advocate. The designer of any “real” system can go the same path in the design of a recoverable system.

We advocate modular top-down design. The system defined at each level is modeled by a Petri-net. In this paper, we only present the top level design of the OS. We assume that the same approach is applied to the other levels.

Suppose that our simplified operating system is described as following:

1. The system loads two jobs, executes them, and only when both are finished it can load two other jobs.
2. The system can be in one of two main states: WAIT or BUSY.
3. In WAIT two jobs can be loaded, and then the system goes to the state BUSY. (Note that in this configuration, the operating system can also execute one job at a time by loading a job and a “null” job.)
4. In the state BUSY, the jobs are executed and when both are finished, the system goes to the state WAIT.
5. After the jobs are loaded, each one is executed in the following steps:
   (1) the operating system starts the job (by setting initial values, allocating resources, etc.),
   (2) the operating system checks if the job is finished. If the job is not finished, the OS executes monitor subroutines as necessary (tests, resource allocation and deallocation, etc.) and goes to the actual execution of the user program (5.3).
   (3) The user program is executed until a “call monitor” is found. Then step 5.2 (above) is executed. These execution steps can be time sliced for the two jobs.
   (4) When the two jobs are completed, the OS goes to the state WAIT.

We note that the previous OS description corresponds to the PN of Figure 1 (or the TM of Figure 2) when the following interpretations are given to the conditions and bars:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = WAIT</td>
<td></td>
</tr>
<tr>
<td>2 = BUSY</td>
<td></td>
</tr>
<tr>
<td>3 = JOBS READY</td>
<td></td>
</tr>
<tr>
<td>4 = OS MONITORING</td>
<td></td>
</tr>
<tr>
<td>5 = JOBS IN EXECUTION</td>
<td></td>
</tr>
<tr>
<td>6 = JOB COMPLETED</td>
<td></td>
</tr>
<tr>
<td>1 = JOB LOADER</td>
<td></td>
</tr>
<tr>
<td>2 = INITIALIZE JOB</td>
<td></td>
</tr>
<tr>
<td>3 = TERMINATION CHECK</td>
<td></td>
</tr>
<tr>
<td>4 = MONITOR SUBROUTINE</td>
<td></td>
</tr>
<tr>
<td>5 = USER PROGRAMS</td>
<td></td>
</tr>
<tr>
<td>6 = CHECK FOR TERMINATION OF TWO JOBS</td>
<td></td>
</tr>
</tbody>
</table>

Each bar denotes a subroutine (or a part of a subroutine), the distribution of tokens among the conditions denotes the state of the system, and the arcs represent the control structure.

Suppose that the user program executes input (or output) and then wait for interrupt. An example of a temporary failure would be that the input device never sends an interrupt request, or that the interrupt request was sent, but because of a failure it was not actually executed. In all these kinds of failures (and others) the user program will remain waiting forever. Thus, bar 5 (Figure 1) will be using the “resources” provided by condition 5, but it will never fire. This situation can be represented as a “loss of token” in condition 5.

The recoverability after “loss of token” in condition 5 was presented in the previous section. As explained there, the process represented in Figure 1 cannot be recoverable if there is not introduced some limitations on the execution times of the associated events. The recoverable TPN was given in Figure 6. Bar 4 was split into four different bars that can fire only if a failure has not occurred. In other words, before the OS allocates a token to condition 5, it checks the system status in order to find out if any token was lost. In case of failure, bar 4 cannot fire and thus bar 3 will fire. After a time long enough (see inequality (1)) the “recovery routine” (bar 7) is activated returning the system to normal initial state. Note that in this case of failure, one or two jobs can be destroyed, but the system recovers and jobs can be executed again.

Note that the system is recoverable only if inequality (1) is satisfied. Thus, the maximal execution time between
successive "monitor calls" from the user programs \((t^5)\) must be finite. The other times appearing in (1) are usually known and are finite.

SUMMARY

We have demonstrated a methodology for designing and checking a system for certain recoverability properties. In order to do this, it was necessary to accept constraints on the execution time of its parts.

We believe that the method presented can be used as a practical tool. In order to apply the method efficiently, the designer is urged to refer himself to the references of this paper for a wider mathematical background of the approach used.

REFERENCES

7. Merlin, Ph. M. and D. J. Farber, A Note on Recoverability of Modular Systems, Department of Information and Computer Science, University of California, Irvine, California 92664, June 1974.