Control Synthesis for a Smart Card Personalization System using Symbolic Model Checking*

Biniam Gebremichael and Frits Vaandrager

Nijmeegs Instituut voor Informatica en Informatiekunde
University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
[biniam,fvaan]@cs.kun.nl

Abstract. Using the Berkeley SMV symbolic model checker we synthesize, under certain error assumptions, a controller for the smart card personalization system, a case study that has been proposed by Cybernetix Recherche in the context of the EU IST project AMETIST. The controller that we synthesize, and of which we prove optimality, has been previously patented. Due to the large number of states (which is beyond 10^{11}), this control synthesis problem appears to be out of the scope of existing tools for controller synthesis, which typically use some form of explicit state enumeration. Our result provides new evidence that model checkers can be useful to tackle industrial sized problems in the area of scheduling and control synthesis.

1 Introduction

Background

Model checking involves analyzing a given model of a system and verifying that this model satisfies some desired properties. System models are typically described as finite transition systems, while properties are described in terms of temporal logic. Once the definition of the system, **S**, and its property, ψ, are fixed, the model checking problem is easily described as $\mathbf{S} \models \psi$? (does **S** satisfy ψ?). Thanks to the symbolic representation of transition systems, state-of-the-art model checking tools are now capable of solving such problems for models with more than 10^{20} states [BC90].

Control synthesis, on the contrary, does not assume the existence of a model of the full system. Instead, it considers the uncontrolled plant and tries to synthesize a controller by finding a possible instance of a

* This work was supported by the European Community Project IST-2001-35304 AMETIST, http://ametist.cs.utwente.nl.

model that satisfies a desired property. Control synthesis for Discrete Event Systems (DES) has been extensively studied over the past two to three decades, and a well-established theory has been developed by Ramadge and Wonham [RW89]. The Ramadge and Wonham framework (RW) is based on the formal (regular) language generated by a finite state machine. The RW plant model P (*generator*) is obtained by describing the plant processes in terms of a formal language which is generated by a finite automaton. A *means of control* is adjoined to this *generator* by identifying the events that can be enabled or disabled by the controlling agent. The specifications S_p are described in terms of formal language generated by P. The controller is then constructed from a recognizer for the specified language given by S_p. An alternative approach is the timed transition model of Ostroff [OS89], where the specification is given as a temporal logic formula instead of a formal regular language.

Control synthesis problems for Discrete Event Systems like the Cybernetix smart card personalization system [Al02] are covered by the Ramadge and Wonham supervisory control theory. In the present paper, however, we solve the problem using a model checker, namely SMV [McM93].[1] This approach allows us to benefit from the (BDD-based) symbolic representation technique of SMV and to solve the problem which, because of its size, would be intractable otherwise. Our results demonstrate that model checkers can be useful to solve problems in the area of scheduling and control synthesis.

Outline

Using SMV we synthesize a controller for a smart card personalization system, which has previously been patented by Cybernetix Recherche. We also show that this controller or scheduler, known as the "super single mode" [Al02] is optimal in the absence of errors. Finally, we synthesize a defective cards treatment that stabilizes the system to the super single mode.

The paper is structured as follows: Section 2 provides a formal definition of the uncontrolled plant of the smart card personalization system, and defines the correctness and optimality criteria. Section 3 explains the super single mode, and how it was generated using SMV. Section 4 deals with systems with faulty cards. We list the errors that may occur during the operations of the machine, show how to deal with such errors, and give

[1] We use the version of SMV developed at Cadence Berkeley Laboratories, see `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`.

an overview of the synthesized error treatment methods. We conclude the paper by pointing out some observations and directions for future work in Section 5. The complete SMV code for the super single mode and defective card treatment is provided in Appendices A and B, respectively. An electronic copy of this code and also of the trace simulator that we developed to visualize the schedules are available via the URL

`http://www.cs.kun.nl/ita/publications/papers/biniam/cyber`.

Related Work

The Ramadge and Wonham framework has been implemented by several research groups and industries. One of the tools developed by Wonham and his research team is CTCT (C based Toy Control Theory)[2], a tool that was basically built for research purposes only, and uses an exhaustive list to represent the model. Its capacity, as the name indicates, has never extended beyond toy examples. A new approach, Vector Discrete Event Systems, was studied in [LW93,LW94] to alleviate the shortcoming of CTCT by exploiting the structural properties of DES. Although this approach resulted in better performance, its structural analysis approach cannot be generalized [CL99].

The UMDES-LIB library [SSLST95] developed by the DES group at the University of Michigan is another implementation of a control synthesis tool, which is very similar to RW supervisory theory. UMDES-LIB is a library of C routines written for the study of discrete event systems modeled by finite-state machines (FSM). There are several routines for the manipulation of FSM, including routines that implement many of the operations of supervisory control theory, and routines that implement part of the methodology developed at University of Michigan for failure diagnosis of discrete event systems.

Bertil Brandin at Siemens, Muenchen, also developed a tool for DES control synthesis, which incorporates heuristics to deal with large systems composed of multiple FSMs [Br96]. Bruce Krogh and his group at Carnegie Mellon University developed a tool for Condition/Event Systems [SK91] which is similar to the supervisory control theory of Ramadge and Wonham. Martine Fabian and Knut Åkesson [AF99] at Chalmers University in Gothenburg, Sreenivas at the University of Illinois at Urbana Champaign [SK91] and several other researchers have also developed similar software.

[2] See `http://odin.control.toronto.edu/people/profs/wonham`.

All the above tools lack symbolic representation of state transitions, and suffer from state space explosion problems. A Binary Decision Diagram (BDD) like data structure called Integer Decision Diagram (IDD) has been used to represent sets of states symbolically. For example, Gunnarsson in [JG97] and Zhang and Wonham in [ZW01] have used IDDs in their implementation. This approach is quite promising for dealing with large systems, but it is still in the laboratory stage, and not available to the public.

Our main motivation for using SMV is thus to overcome this deficiency and benefit from symbolic representation of SMV. The smart card personalization system is quite a large system and cannot be handled with a tool that does not use symbolic representation. Our paper shows how the control synthesis can be solved using a model checker and presents new evidence that model checkers can be useful in solving problems in the area of scheduling and synthesis.

We were the first to model the smart card personalization system and to synthesize a controller for it. However, the same case study has also been addressed by other members of the AMETIST consortium. T. Krilavicius and Y. Usenko [KU03] constructed models using UPPAAL and μCRL, and used these to synthesize controllers. Whereas in our model production of cards is essentially an infinite process, Krilavicius and Usenko only consider scheduling of a finite number of cards. As a consequence, they do not synthesize the super single mode. Inspired by [KU03], T. Ruys used SPIN to synthesize a controller for the smart card personalization machine [Ru03]. Also this model only considers scheduling of a finite number of cards (the largest parameter values considered are 5 cards and 4 stations). In order to handle the state space explosion, Ruys encodes branch & bound search strategies in SPIN. In addition, he has to instruct SPIN to use a number of heuristics, which in our view are both complex (the code for the heuristics is longer than the code of our entire model!) and debatable (Ruys assumes that cards cannot overtake each other; in the real machine this is possible with the help of the personalization stations). A. Mader in [Ma03] applied decomposition and mixed strategies to model and synthesis a controller for the extended smart card personalization machine that include printers and flippers. G. Weiss employed Life Sequence Charts (LSC) to synthesize a scheduler with smart play-in/play-out approach [We03]. None of the mentioned approaches deals with error handling.

2 Smart Card Personalization System

The "smart card personalization system" is a case study that has been proposed by Cybernetix Recherche in the context of the EU IST project AMETIST [Al02]. The case study concerns a machine for smart card personalization, which takes piles of blank smart cards as raw material, programs them with personalized data, prints them and tests them.

The machine has a throughput of approximately 6000 cards per hour. It is required that the output of cards occurs in a predefined order. Unfortunately, some cards may turn out to be defective and have to be discarded, but without changing the output order of personalized cards. Decisions on how to reorganize the flow of cards must be taken within fractions of a second, as no production time is to be lost.

The goal of the case study is to model the desired production requirements as well as the timing requirements of operations of the machine, and on this basis synthesize the coordination of the tracking of defective cards. More specifically, the goal is to synthesize optimal schedules for the personalization machine in which defective cards are dealt with, i.e., schedules in which

1. cards are produced in the right order (safety),
2. throughput is maximal (liveness).

2.1 The Uncontrolled Plant Model

Figure 1 shows a simplified smart card personalization machine. The ma-

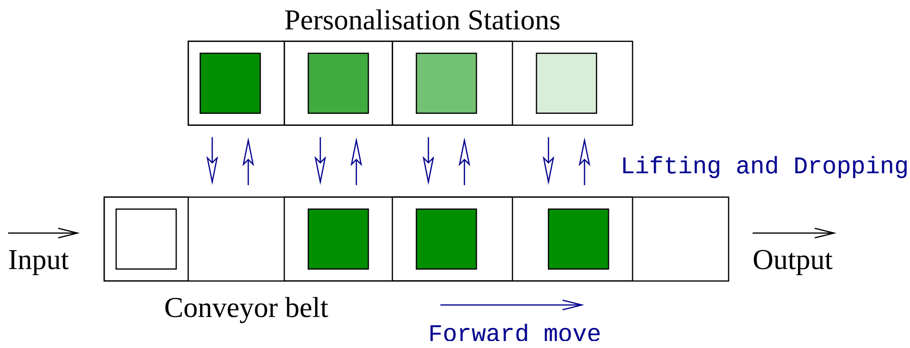

Fig. 1. Simplified smart card personalization machine.

chine consist of a conveyor belt and personalization stations mounted on top of it. The machine also has an input station and an output station, which are situated on the left and right side of the belt respectively. New cards enter the system through the input station and advance to the right one step at a time. At some point, a card is lifted up to one of the personalization stations, spends some time there (is personalized), and is then dropped back onto the belt. The card then moves towards the output station for testing and delivery. The actual machine is considerably more complicated than the machine in Figure 1, but our aim is to find a scheduler that effectively utilizes the personalization stations and optimizes throughput. The simplified model of the machine appears to be adequate for this purpose.

The SMV model for the uncontrolled machine is a collection of processes running concurrently: `forward` (moving a belt one step to the right) and, for each personalization station j, $\texttt{lift\_drop}_j$ (lifting/dropping a card from/to the belt to/from station j). We employ a discrete model of time, in which one time unit is equivalent to one forward move of the belt. All personalization stations are identical and need `S` time units to personalize a card. We assume lifting and dropping takes no time.

We assume there are `M` stations (denoted by $\texttt{b}_j$), and `N = M+2` slots in the belt (denoted by $\texttt{a}_j$) as shown in Figure 2. To make model checking possible, the number of different personalizations is assumed to be bounded by some value `K`, which is a multiple of `M`. Each slot or station will have a value as shown in Table 1. An empty slot/station is coded twice as (`-3` and `-2`) in order to distinguish between the initial value (`-3`) and the slot/station being emptied along the way (`-2`). This allows us to control intermediate blank slots more efficiently, as will be explained below. We also use an integer variable $\texttt{x}_j$, ($\texttt{0} \leq j$ `<` `M`) as a clock to record how long a card has been held in station j.

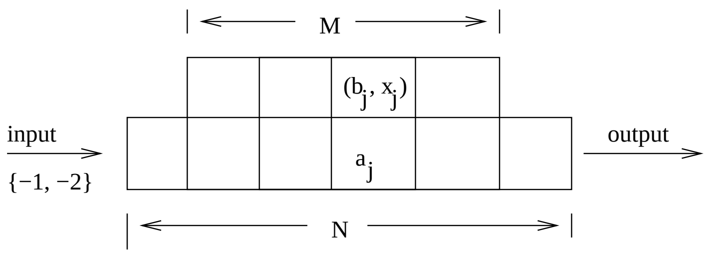

Fig. 2. The model of the smart card personalization machine.

parameter	represents
M	number of stations
N	total number of slots
K	different number of personalizations
S	time needed for personalization

slot/station value	meaning
-3	empty (initial value)
-2	emptied
-1	new card
j, $0 \leq$ j $<$ K	personalized with j
K	defective card

Table 1. System parameters and encoding of values.

Formally, the process `forward` is defined as follows (for a complete specification of the system we refer the reader to Appendix A.).

```
module forward(a,b,x){

  next(a[0]):={-1,-2};
  for(j=1;j<=N-1;j=j+1)      next(a[j]):=a[j-1];
  for(j=0;j<=M-1;j=j+1){
     if(x[j]<S & b[j]>=0)   next(x[j]):= x[j]+1;
  }
}
```

Module `forward` specifies a process that runs concurrently with the rest of the processes in the system. New cards enter the system from the left end of the conveyor belt with every forward move of the belt: `a[0]` can either receive a new card (`-1`) or no card (`-2`). The other slots in the conveyor belt (`a[i], i > 0`) get the value of the previous slot. Clocks of the busy stations advance by one (`next(x[j]):= x[j]+1`), until they reach the maximum value (`S`), after which they remain unchanged.

Process `lift_drop`$_j$ is defined as follows.

```
module lift_drop(a,b,x,j){

  if(b[j] <= -2 & a[j+1] = -1){
    next(b[j]):= 0..K;
    next(a[j+1]):=b[j];
    next(x[j]):=0;
  }

  else if(b[j] >= 0 & a[j+1] = -2 & x[j] = S){
    next(b[j]):= -2;
```

```
    next(a[j+1]):=b[j];
    next(x[j]):= 0;
  }
}
```

There are as many `lift_drop` processes as there are personalization stations, and they all run concurrently. When a `lift_drop` process is running, one of the following three cases may occur:

1. The associated station is idle (`b[j]` <= `-2`) and the slot beneath contains a card that is not personalized (`a[j+1] = -1`). In this case the card is lifted, i.e., the process
   – nondeterministically assigns a personalization value to the lifted card (`next(b[j]):= 0..K`);
   – resets the value of the slot to the previous value of the station (`next(a[j+1]):=b[j]`). This previous value is `-3` if this is the first time it has personalized a card and `-2` otherwise;
   – resets the local clock to zero (`next(x[j]):=0`).
2. The associated station contains a fully personalized card (`b[j]` >= `0` & `x[j] = S`) and the slot underneath is empty (`a[j+1] = -2`). In this case the card is dropped, i.e., the process
   – sets the station to idle state (`next(b[j]):= -2`);
   – drops the card into the slot underneath (`next(a[j+1]):=b[j]`);
   – resets the local clock to zero (`next(x[j]):=0`).
3. If none of the above two conditions is met, running the lift_drop process will have no effect.

Correctness The desired correctness property is:

> *There exists a run that always produces personalized cards in the right order.*

To formalize the concept of "right order", an observer process is introduced that compares the output value with the expected value. Formally, the observer is defined as follows. We introduce a new state variable `out`, which initially is `0` and assume `K` is a multiple of `M`, say `2.M`. The behavior of the observer is specified by:

```
if(out = a[N-1])       next(out):= (out+1) mod K;
  else if(a[N-1]>-2)  next(out):= K;
```

If cards are not produced in the right order or if a card is output that has not been personalized, the observer sets the value of `out` to the "error" value K. The control objective then becomes to ensure that the observer will never detect an error. We can synthesize a controller that realizes this (if it exists) by asking SMV whether the following CTL formula holds:

$$\texttt{AF}\neg(\texttt{out} < \texttt{K}). \tag{1}$$

If this formula does *not* hold then there exists an infinite run in which for all states `out < K`, i.e., the observer never detects an error. In this case SMV will provide a counter example, which essentially is an infinite schedule for the machine that meets the control objective.

Optimization Obviously, there are many runs in which all states satisfy `out < K`, for instance, a run in which the machine produces no cards at all. The interesting runs are those with high throughput. Using SMV we were able to establish (for small values of M) that the maximal throughput that can be obtained is bounded by "(`t.M + t`) time units for `t.M` cards". That is, if we want to produce `t.M` cards at least `t.M + t` time units are required after the first personalized card arrives in $\texttt{a}_{N-1}$, or `t.M + t + S + N` if we include the leading empty slots. In this formula, `t` corresponds to a blank slots we need in between. To see why such a bound is necessary, take the leftmost personalization station. This station drops the personalized card into a slot which was originally blank. Thus for this station to produce `t` cards, it needs `t` blank slots.

To minimize the blank slots in the output and in order to guide SMV towards optimal schedules, we introduce the "blank tolerance condition" of the machine, a new state variable `tl`, which is initially `0`, and is incremented and decremented as follows:

```
if(a[N-1]=-2)                                 next(tl):=tl-1;
else if( a[N-1]>=0 & (a[N-1] mod S) = S-1)  next(tl):=tl+1;
```

We add `1` to `tl` each time `S` cards have been produced (a_{N-1} `modulo S = S-1`). We decrement `tl` with `1` whenever a blank slot arrives ($\texttt{a}_{N-1}$ `= -2`). However, we start decrementing only after the leading blank slots (`a[N-1] = -3`) have passed. In all other cases we leave the value of `tl` unchanged.

Now we ask SMV whether the following CTL formula holds:

$$\texttt{AF}\neg(\texttt{out} < \texttt{K} \wedge \texttt{tl} \geq \texttt{0}). \tag{2}$$

If this formula does not hold, there exists an infinite scheduler that maintains the invariant $\texttt{tl} \geq \texttt{0}$. This means that each time when the system has produced `S` cards, the observer tolerates a single blank slot.

3 The Super Single Mode

Using the approach outline in the previous section, the example run in Table 2 was generated. With a "normal-speed" PC we were able to generate example runs for `M` $\leq$ `4`. The runs exhibit the schedule of the super single mode as patented by Cybernetix. Table 2 shows the first 19 configurations of the smart card personalization machine with `M = 4`, `S = 4`, `K = 12`. Each row represents a single configuration at a given time. The upper part of the row shows the values of the stations, while the lower part shows the values of the slots in the conveyor belt. An empty cell means the slot or the station is idle, a box ($\square$) represents a new card, and a number represents the personalization value of the card contained in the station or in the slot. Table 2 can be read as:

- time 0: the machine is empty.
- time 1: first new card arrives on the conveyor belt.
- time 2: the first card is lifted to station `0`.
- time 4: the second card is lifted to station `1` and it continues likewise.
- time 5: there is no card from the input.
- time 6: station `0` finishes personalizing a card with value `0`. In super single mode, `M` (`4` in this example) time units are required to personalize a card.
- time 7: station `0` proceeds with personalizing another card with a different value (namely `4`). Note that value `3` is not taken yet. This pattern shows that the order of output is exactly the same as the order of the cards when they are fed into the machine, but the production order is different, and there is an overlap between rounds. This overlap is even more clearly visible when a machine with `8` (instead of `4`) personalization stations is considered.

If in our model a station is allowed to take more than `M` time units for personalizing a card, i.e., `S > M`, then CTL formula (2) holds. In other words: if the conveyor belt is rolling faster than the personalization stations can handle then personalizing `M` consecutive cards becomes impossible.

Similarly, for a personalization time of `M` time units, if we have `M+1` consecutive new cards followed by empty slots (even with lots of empty slots), then it becomes impossible to personalize all of them. This result implies that the super single mode is optimal in the absence of errors.

time	input	station 0	station 1	station 2	station 3	output
0						
1	□					
		0				
2	□					
		0				
3	□	□				
		0	1			
4	□	□				
		0	1			
5		□	□			
			1	2		
6	□	0	□			
		4	1	2		
7	□		0	□		
		4		2	3	
8	□	□	1	0		
		4	5	2	3	
9	□	□		1	0	
		4	5		3	
10		□	□	2	1	0
			5	6	3	
11	□	4	□		2	1
		8	5	6		
12	□		4	□	3	2
		8		6	7	
13	□	□	5	4		3
		8	9	6	7	
14	□	□		5	4	
		8	9		7	
15		□	□	6	5	4
			9	10	7	
16	□	8	□		6	5
		12	9	10		
17	□		8	□	7	6
		12		10	11	
18	□	□	9	8		7

Table 2. The super single mode for 4 personalization stations.

4 Error Recovery

The control objective for the smart card personalization machine is to personalize cards in the right order even in the presence of errors. The super single mode, as explained above, only works for a perfect machine that makes no errors. In general, it is difficult to prevent errors from occurring (even though they errors are rare, appr. 1 in 6000 cards), and so it makes our approach more realistic if we allow for the occurrence of errors in our model, and provide a means of recovering from them.

There are several methods to achieve fault-tolerant behavior. Our approach is inspired by the concept of *self-stabilization* [Dij74,Te94], which is well-known from the area of distributed algorithms. An algorithm is called stabilizing if it eventually starts to behave correctly (i.e., according to the specification of the algorithm), regardless of the initial configuration.

Figure 3 shows the production cycle of the personalization machine under the super single mode. In the normal mode of operation the machine loops on the super single mode cycle (the continuous line). This loop is also shown in Table 2 with actual figures. The configurations of the machine at time 9, 10, 11, 12, 13 are equivalent (personalization value modulo `M = 4`) to the configurations at time 14, 15, 16, 17 and 18 respectively. Thus the super single mode enters the loop at time 9 and loops forever with a period of 5 time units.

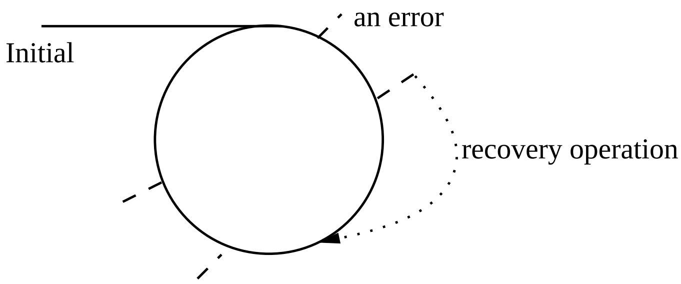

Fig. 3. Stabilization of the smart card personalization system.

However, when an error occurs (dashed line in figure 3), an error recovery treatment (dotted line) should be conducted to stabilize the system and bring it back to the loop. We use SMV to synthesize the error recovery treatment that brings the machine back to the loop. Basically, our approach is as follows:

1. Use SMV to synthesize a regular super single mode run, as described in the previous section.
2. Pick a state on this run and manually introduce an error; the new error state s now becomes the start state of the model.
3. Pick an arbitrary state t on the super single mode cycle, and encode this as an SMV state formula φ.
4. Ask SMV whether the following formula holds

$$\mathtt{AG}\neg\varphi. \tag{3}$$

If formula (3) does not hold then SMV generates a counterexample; this counterexample is the schedule for a recovery operation that brings the system from state s back into super single mode.

Note that, unlike the theory of self-stabilization, we do not consider arbitrary initial configurations, but only configurations that have been obtained by introducing a single error into a super single mode configuration.

4.1 Types of Errors

It is easy to list many scenarios that can make the system behave erratically. In this paper we will only consider errors that may occur in the card. That is:

1. Type 1 errors (E1) are errors in a smart card originated from physical damage or other reasons. This type of errors is detected by the personalization stations. In Table 3, as an example, personalization station 6 detected an error on a card at time 14.
2. Type 2 error (E2) are errors originating from the personalization station when cards are personalized wrongly, which makes them unusable. This type of errors is detected by a tester situated next to the personalization stations. Table 3 shows another error of this type at time 22. The card personalized with data 4 is found to be defective when tested.

To make our system recoverable from these errors, we will modify our model in two ways: by adding extra operations and by expanding the belt in both directions.

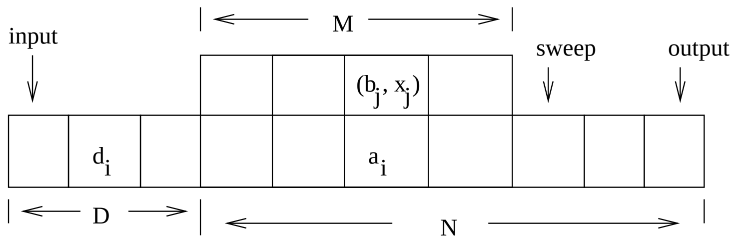

Fig. 4. Expanded model of the smart card personalization machine.

time	input	personalization stations								output
		0	1	2	3	4	5	6	7	(tester)
9										
10	□					4				
11	□	8								
12	□						5			
13	□		9							
14	□							E1		
15	□			10?						
16	□								7?	
17	□				11?					
18										0
19	□					12?				1
20	□	16?								2
21	□						13?			3
22	□		17?							E2
23	□							14?		5
24	□			18?						E1
25	□								15?	7?
26	□				19?					
27										8

Table 3. The super single mode for 8 personalization stations with error.

4.2 Recovery Operations

If a defective card is detected in the tester then, in order to maintain correctness (i.e., produce personalized cards in the right order), the defective card has to be removed, a replacement card has to be produced, and inserted in the right position. In order to realize this, first the defective card has to be swept off the belt, and then the belt has to go back to one of the personalization stations to insert the replacement card in the right position. For these purpose we enrich our model with 'backward' and 'sweep' operations.

The backward move is the same as the forward move except that it moves the belt in the opposite direction. The forward move is the "normal" way of moving the belt, the backward move is used only to handle defective cards [Al02]. We assume that a backward move takes 1 time unit per step. The formal definition is:

```
module backward(d,a,b,c){

  if(d[0]<0){
    next(a[N-1]):= -2;
```

```
    next(d[D-1]):=a[0];

    for(j=N-2;j>=0;j=j-1)       next(a[j]):=a[j+1];
    for(j=D-2;j>=0;j=j-1)       next(d[j]):=d[j+1];

    for(j=0;j<=M-1;j=j+1)
       if(x[j]<S & b[j]>-2)    next(x[j]):= x[j]+1;
  }
}
```

When the belt moves backward, the leftmost cards on the belt are also pushed back to the edge. For technical reasons explained in [Al02], the preferred way of treatment is to expand the belt to the left. As shown in Figure 4, the gap between the input station and the first personalization station, denoted by $\mathtt{d}_i$ `(0` $\leq$ `i` $\leq$ `D, D = N-1)`, is important for backward movement. Similarly, the belt is also expanded to the right: `N` (= `M+2`) covers the extended slots in the right side.

A sweeper is a device that kicks defective cards from the belt. In the physical machine, a sweeper is situated after the personalization station. Formally the sweep operation is defined by:

```
module sweep(a){
  if(a[M]=K)     next(a[M]):=-2;
}
```

4.3 Safety Requirements

During the stabilization process, the machine executes operations that are not performed in super single mode. Even if the machine is allowed to perform these special operations, there are some safety requirements that have to be obeyed by the control program. These are shown in Table 4. Observe that the `forward` and `sweep` operations explained above are guarded with `d[0] < 0` and `a[M] = K` respectively. The `forward` operation is also modified and guarded with `(a[N-1] = out | a[N-1] = -2)`. The complete SMV code for error recovery treatment is given in Appendix B.

4.4 Results

We were able to prove that the convergence property holds when considering a single error. As shown by the example runs generated, error treatments are different for different types of errors, and they also depend

Operation	Safety requirements	meaning
backward	`d`$_0$ `< 0`	no processed card reaches input station, unprocessed (new) cards can return back to the input station
forward	`a`$_{N-1}$ `= out` $\vee$ `a`$_{N-1}$ `= -2`	no unexpected card reaches the tester station
sweep	`a`$_M$ `= K`	only defective cards are swept

Table 4. Safety requirements for belt operations.

on which station the error occurs and on which station the replacement card has to be produced. To illustrate these observation we present three examples. The examples have been generated by SMV. We have no proof of their optimality: there could exist better treatment mechanisms, that we do not know about.

Example 1. (Defective card treatment of type 1) Once a station detects that a card is defective, the personalization value is handed over to the next station. Table 5 shows how an error in station 1 is treated. Station

time	input	personalization stations								output
		0	1	2	3	4	5	6	7	
11	□	8								
12	□						5			
13	□		(E1)							
14	□							6		
15	□			9						
27		24								8
28	□					20				
19	□		25							9

Table 5. Error recovery by skipping personalization values.

2 produces a card with personalization value 9 instead of 10, and station 1 keeps the defective card as if no error occurred.

Example 2. (Defective card treatment of type 2) When a card is detected as defective later in the tester, a replacement card is produced by the station that is currently available. Table 6 shows how such an error is treated. The belt moves backward, the new card is placed in its right place, and is forwarded to the tester again. In Table 6, at time 24 the 6^{th}

time	input	personalization stations								tester
		0	1	2	3	4	5	6	7	
12	□						5			
21	□						13			3
22	□		17							4
23	□							14		5
24	□			6						(E2)
25*	□	□		10	9	8	□	7		
26*	□	10	9	8	□	7				
32*	□	8	7	6						
32	□			18						6

Table 6. Error recovery by backward and replace. (*The columns marked with * represent the value of the associated slots in the conveyor belt.*)

card is found defective and a replacement card is produced by station 2. The belt moves backward and the 6^{th} card is placed at front. After 8 time units, the new card is forwarded and tested again.

Example 3. (Defective card treatment of type 2) Consider the following scenario, where type 2 error is detected in the tester and the available station is in the second half (stations numbered $\frac{M}{2}$ up to $M - 1$).

As shown in Table 7, at time 23 the 5^{th} card is found defective and the station 6 is ready. If a replacement card would be produced in this

time	input	personalization stations								tester
		0	1	2	3	4	5	6	7	
12	□						5			
21	□						13			3
22	□		17							4
23	□							14		(E2)
24	□			5						6
25*	□	□		10	9	8	□	7	6	
26*	□		10	9	8	□	7	6		
32*	□	7	6	5						
32	□							14		5

Table 7. Defective card treatment for error type 2.

station, then personalization number 14 would be skipped. But this would introduce another error, because the 16^{th} and 17^{th} cards are already in preparation and they can not be altered. Instead we can produce the card in the next station that becomes available, which is station number 2.

For type 1 errors, the treatment is similar if the error arises in the second half stations. Thus, this trace suggests the second half stations may not be used to produce replacement cards.

4.5 Cost of Error Recovery

An upper bound on the number of time units spent recovering from an error can be calculated as follows.

1. Once an error is detected by the tester, one step forward may be necessary if it is an error like in Example 3.
2. To reproduce a replacement card we will require `S = M` time units, during this time the belt rolls back to the station.
3. Once the card is reproduced, it will take another `M` time units for the new card to reach the tester. In practice the belt can move forward faster than `M` time units, and the time spent to reach the tester will be smaller.

Thus `2.M + 1` time units are required in the worst case to recover from a single error. It is possible to tighten this upper bound by introducing fast forward and fast backward moves.

5 Conclusions

Using SMV, we rediscovered the super single mode that has previously been patented by Cybernetix. This result gives us a new evidence that model checking can also be useful as a design aid for new machines. Our approach also allowed us to generate defective card treatments, that may arise due to damaged cards and wrong personalization.

The input language of Cadence SMV is sufficiently expressive to encode in a natural and compact way a simplified model of the personalization machine. However, safety and liveness properties for multiple error treatments (of single or multiple types) are complicated to express in temporal logic, especially when dealing with the uncontrolled plant. Nevertheless, by decreasing the degree of uncontrollability of the plant, we believe multiple errors can be handled and more complex discrete time models of the actual Cybernetix design (including the controller) can be described.

A possible disadvantage of our approach is that the SMV descriptions are difficult to understand for people who are not familiar with formal methods (unlike say Petri nets). However, a clear advantage is that our description can serve directly as input for a powerful model checker.

References

[AF99] K. Åkesson and M. Fabian. Implementing Supervisory Control for Chemical Batch Processes. *Proc. of IEEE Int. Conf. on Control Applications, pages 1272-1277, Hawaii, USA* 1999.

[Al02] S. Albert. CYBERNETIX Case Study Informal Description. *Advanced Methods for Timed Systems,* 2002. See `http://ametist.cs.utwente.nl`

[BC90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* 1990.

[Br96] B. A. Brandin. The Real-Time Supervisory Control of an Experimental Manufacturing Cell, *IEEE Transactions on Robotics and Automation, Vol., 12, No.1., pp. 1- 13* 1996.

[CL99] C. G. Cassandras, S. Lafortune. Introduction to Discrete Event Systems *Luwer Academic Publishers, ISBN 0-7923-8609-4.* 1999.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications ACM vol 17. pages 643-644.* 1974.

[JG97] J. Gunnarsson. Symbolic Methods and Tools for Discrete Event Dynamic Systems. *Linköping Studies in Science and Technology, Dissertation.* 1997

[KU03] Tomas Krilavicius, Yaroslav Usenko. Smart Card Personalisation Machine in UPPAAL and μCRL. 2003. In preparation.

[LW93] Y. Li and W.M. Wonham. Control of vector discrete-event systems I - the base model. *IEEE Trans. Autom. Control, 38(8):1214-1227,* August 1993.

[LW94] Y. Li and W.M. Wonham. Control of vector discrete-event systems II - controller synthesis. *IEEE Trans. Autom. Control, 39(3):512-531,* March 1994.

[Ma03] A. Mader. Cybernetix case study: approaching scheduling by decomposition and mixed strategies. 2003. In preparation.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[OS89] Ostroff, J.S. Synthesis of Controllers for real-time discrete event systems. *IEEE Proceedings of the 28th Conference on Decision and Control,* Tampa, Florida, 1989.

[Ru03] T. C. Ruys. Optimal Scheduling using Branch and Bound with SPIN 4.0. To appear in the *Proceedings of SPIN 2003, the 10th SPIN Workshop*, Portland, Oregon, USA, May 9-10, 2003.

[RW89] P.J. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE, 77(1):* 81–98, 1989.

[SK91] R.S. Sreenivas and B.H. Krogh. On Condition/Event Systems with Discrete State Realizations. In *Discrete Event Dynamic Systems. Theory and Applications 1, pages 209-236. Flumer Academic, Boston, USA* 1991.

[SSLST95] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, September 1995.

[Te94] G. Tel. Introduction to Distributed Algorithms. *Cambridge University Press.* 1994.

[We03] G. Weiss. LSCs modeling and Cybernetix CS. 2003. In preparation.

[ZW01] Z. Zhang and W.M. Wonham. STCT: An Efficient Algorithm for Supervisory Control Design. *ECE Department, University of Toronto* 2001.

Appendix A

SMV code for the uncontrolled plant of the simplified smart card personalization system.

```
#define M 4     /* Number of personalization stations   *
                 * Use 3 or 2 for fast output           */
#define N M+2   /* The slots of the conveyor belt       */
#define K M     /* Different personalizations           */
#define S M     /* Personalization time                 */

module forward(a,b,x,out,tl){

  /* a slot is initially empty or contains new card */
  next(a[0]):={-1,-2};

  /* the belt advances one step forward */
  for(j=1;j<=N-1;j=j+1)    next(a[j]):=a[j-1];

  /* clock x advances */
  for(j=0;j<=M-1;j=j+1){
    if(x[j]<S & b[j]>=0)
      next(x[j]):= x[j]+1;
  }

  /* checking order of production */
 if(out = a[N-1])     next(out):= (out+1) mod K;
 else if(a[N-1]>-2)  next(out):= K;

 /* blank tolerance check */
 if(a[N-1]=-2)   next(tl):=tl-1;
 else if( a[N-1]>=0 & (a[N-1] mod S) = S-1)  next(tl):=tl+1;

  FAIRNESS running;
}

module lift_drop(a,b,x,j){

  if(b[j] <= -2 & a[j+1] = -1){ /* lift */
    next(a[j+1]):=b[j];         /* the slot is emptied */
    next(b[j]):= 0..K;          /* assign a personalization value */
```

```
    next(x[j]):=0;              /* clock reset */
  }
  else if(b[j] >= 0 & a[j+1] = -2 & x[j] = S){ /* drop */
    next(b[j]):= -2;            /* reset station */
    next(a[j+1]):=b[j];         /* drop card and the corresponding
   slot receives the personalized card */
    next(x[j]):=0;              /* clock reset */
  }
FAIRNESS running;
}

module main(){
 a: array 0..N-1 of -3..K;
 b: array 0..M-1 of -3..K;
 x: array 0..M-1 of -2..S;
 out: -2..K;
 tl: -1..1;

 tk: process forward(a,b,x,out,tl);

 for(j=0;j<=M-1;j=j+1)   ld[j]:process lift_drop(a,b,x,j);

 for(j=0;j<=N-1;j=j+1)   init(a[j]):=-3;

 for(j=0;j<=M-1;j=j+1){  init(b[j]):=-3;
                         init(x[j]):=0;
   }

 init(out):=0;  init(tl):=0;

 correctness:  assert F ~((out < K) & (tl>=0));

 FAIRNESS running;
}
```

Appendix B

SMV code for the simplified smart card personalization station and its defect card treatment.

```
#define M 4     /* Personalization stations */
#define K M     /* number of different personalizations */
#define N M+2   /* number of slots on the middle and right side */
#define D M     /* number of slots on the left side */
#define S M     /* speed of personalization */
#define B N

module forward(d,a,b,c,out){

  if(a[M]<K & (a[N-1]=-2 | a[N-1]=out)){

    next(d[0]):= {-2,-1};

    next(a[0]):=d[D-1];

    for(j=1;j<=N-1;j=j+1)          next(a[j]):=a[j-1];
    for(j=1;j<=D-1;j=j+1)          next(d[j]):=d[j-1];

    for(j=0;j<=M-1;j=j+1){         next(b[j]):=b[j];

    if(c[j]<S & b[j]>=0)         next(c[j]):= c[j]+1;
    else                         next(c[j]):= c[j];
    }

    if(out = a[N-1])               next(out):= (out+1) mod K;
    else  if(a[N-1] <= -2)         next(out):= out;
    else  next(out):= K;
  }

  else{
    if(a[M]=K) next(a[M]):= -2;
  }

  FAIRNESS running;
}
```

```
module backward(d,a,b,c,bk){
  if(d[0]<0 & bk < B-1){
    next(a[N-1]):= -2;
    next(d[D-1]):=a[0];

    next(bk):= bk+1;

    for(j=N-2;j>=0;j=j-1)           next(a[j]):=a[j+1];
    for(j=D-2;j>=0;j=j-1)           next(d[j]):=d[j+1];

    for(j=0;j<=M-1;j=j+1){          next(b[j]):=b[j];

      if(c[j]<S & b[j]>-2)          next(c[j]):= c[j]+1;
      else                          next(c[j]):= c[j];
    }
  }
  FAIRNESS running;
}

module lift_drop(a,b,c,j){
  if(b[j] = -2 & a[j] = -1){
    next(b[j]):= 0..K;
    next(a[j]):=-2;
    next(c[j]):=0;
  }
  else if(b[j] >= 0 & a[j] = -2 & c[j] = S){
    next(b[j]):= -2;
    next(a[j]):=b[j];
    next(c[j]):= 0;
  }
  FAIRNESS running;
}

module main(){
 d: array 0..D-1 of -2..K; /* slots on the left side */
 a: array 0..N-1 of -2..K; /* slots on the middle and right side */
 b: array 0..M-1 of -2..K; /* personalizations */
 c: array 0..M-1 of 0..S;  /* personalization time counter */
 out: 0..K;                /* observer for the order of cards */
 bk: 0..B;
```

```
 fward: process  forward(d,a,b,c,out);
 bward: process backward(d,a,b,c,bk);

 for(j=0;j<=M-1;j=j+1)
   ld[j]:process lift_drop(a,b,c,j);

 init(a[0]):= -1; init(a[1]):= -2; init(a[2]):= 1;  init(a[3]):= -0;
 init(a[4]):= -2;  init(a[5]):= 3;
 init(b[0]):= 0; init(b[1]):= 4; init(b[2]):= 2; init(b[3]):= 3;
 init(c[0]):= 2; init(c[1]):= 0; init(c[2]):= 3; init(c[3]):= 1;
 init(d[0]):= -1; init(d[1]):= -1; init(d[2]):= -2; init(d[3]):= -1;
 init(out):= 3;  init(bk):=0;

 sft: SPEC AG ~(
a[0] = -1 &  a[1] = -2 &  a[2] = 1 &   a[3] = -0 &
a[4] = -2 &   a[5] = 3 &
b[0] = 0 &  b[1] = 1 &  b[2] = 2 &  b[3] = 3 &
c[0] = 2 &  c[1] = 0 &  c[2] = 3 &  c[3] = 1 &
d[0] = -1 &  d[1] = -1 &  d[2] = -2 &  d[3] = -1 &
out = 3
);

 FAIRNESS running;
}
```