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

Biniam Gebremichael and Frits Vaandrager

Nijmegen Institute for Computing and Information Sciences
University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
[biniam,fvaan]@cs.kun.nl

Abstract. Using the Cadence SMV symbolic model checker we synthesize, under certain error assumptions, a scheduler 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 scheduler that we synthesize, and of which we prove optimality, has been previously patented. Due to the large number of states (which is beyond 10^{13}), this 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, $\mathbf{S}$, and its property, ψ, are fixed, the model checking problem is easily described as $\mathbf{S} \models \psi$? (does $\mathbf{S}$ 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 [4].

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 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 [16]. The Ramadge and

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

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.

In this paper, we consider a problem which in theory could very well be solved using the Ramadge and Wonham supervisory control theory. However, given the size of the state space involved, existing control synthesis tools are (to the best of our knowledge) unable to actually compute a solution. Therefore, instead, we tackled the problem using the symbolic model checker SMV [14].[1] This approach allows us to benefit from the (BDD-based) symbolic representation technique of SMV and to (partially!) solve a problem which, because of its size (more than 10^{13} states), 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 scheduler for a smart card personalization system, which has previously been patented by Cybernetix Recherche. We also show that this scheduler, known as the "super single mode" [2] is optimal in the absence of errors. Finally, we synthesize a set of schedulers for defective card treatment that stabilize the system back to the super single mode. Together, these schedulers constitute a controller for the system under the assumption that a certain amount of time elapses between faults.

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 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.

A full version of this paper appeared as [8]. An electronic copy of SMV code and also of the trace simulator that we developed to visualize schedules are available via the URL

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

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

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 [12, 22] 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 [5].

Other notable developments on this area are: The UMDES-LIB library from University of Michigan [18], Bertil Brandin's tool for DES control synthesis with heuristics [3], a tool for Condition/Event Systems [19], other tool by Martine Fabian and Knut Åkesson [1].

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 [9] and Zhang and Wonham in [23] have used IDDs in their implementation. This approach is quite promising for dealing with large systems, but it is still in 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 scheduler 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. Our work has been inspired by similar approaches that were employed in [7, 10, 15] to synthesize schedulers for industrial size problems.

We were the first to model the smart card personalization system and to synthesize a scheduler for it. However, the same case study has also been addressed by other members of the AMETIST consortium. T. Krilavicius and Y. Usenko [11] 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 [11], T. Ruys used SPIN to synthesize a controller for the smart card personalization machine [17]. 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

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

assumes that cards cannot overtake each other; in the real machine this is possible with the help of the personalization stations). A. Mader in [13] applied decomposition and mixed strategies to model and synthesize 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 [21]. 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 [2]. 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). The order of cards is important as no other sorting mechanism should exist in the system,
2. throughput is maximal (liveness).

2.1 The Uncontrolled Plant Model

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

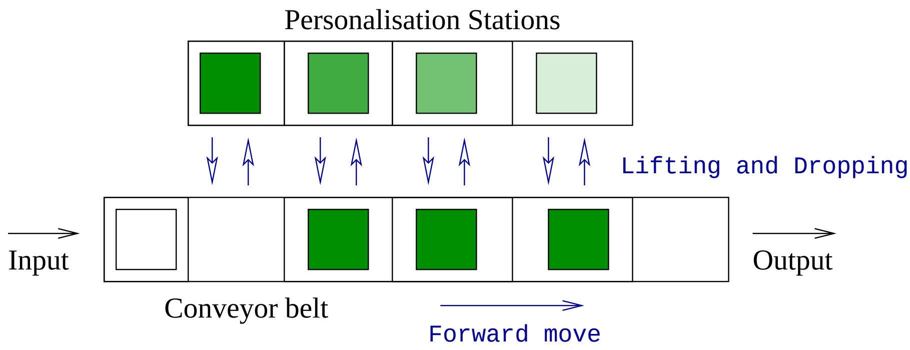

Fig. 1. Simplified smart card personalization machine.

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

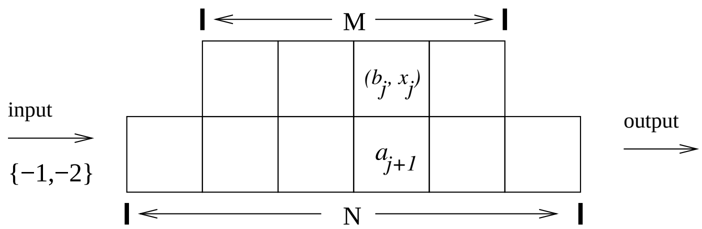

Fig. 2. The model of the smart card personalization machine.

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.

Table 1. System parameters and encoding of values.

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, $0 \le$ j $<$ K	personalized with j
K	defective card

Formally, the process `forward` is defined as follows (for a complete specification of the system we refer the reader to [8]).

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

  next(a[0]):={-1,-2};            /* a new card appears    */
  for(j=1;j<=N-1;j=j+1)           /*non-deterministicaly   */
         next(a[j]):=a[j-1];      /* move the belt forward */
  for(j=0;j<=M-1;j=j+1){
     if(x[j]<S & b[j]>=0)         /* increment clocks of   */
      next(x[j]):= x[j]+1;        /* the busy stations     */
  }
}
```

and the processes $\texttt{lift\_drop}_j$ ($0 \le j <$ M) are defined as:

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

if(b[j] <= -2 & a[j+1] = -1){ /* idle station and new card*/
  next(b[j]):= 0..K;       /* generate a  personalization */
  next(a[j+1]):=b[j];      /* reset the slot              */
  next(x[j]):=0;           /* reset the clock             */
}
else if(b[j] >= 0 &  x[j] = S /* card personalized        */
      & a[j+1] = -2 ){        /* a blank slot beneath    */
  next(a[j+1]):=b[j];         /* drop the card            */
  next(b[j]):= -2;            /* reset the station        */
 }
}
```

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 scheduler that realizes this (if it exists) by asking SMV whether the following CTL formula holds:

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

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, or more specifically with less number of blank slots in the output.

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, in the form of 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 ($\mathtt{a}_{N-1}$ `= -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:

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

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

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

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

3 The Super Single Mode

Using the approach outlined 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` $\leq$ `5` (in the real machine `M` could be `8,16` or `32`). The runs exhibit the schedule of the super single mode as patented by Cybernetix. Table 2 shows the first 19 configurations of the the super single mode 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 (□) 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.

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 errors are rare, approximately 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* [6, 20], 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

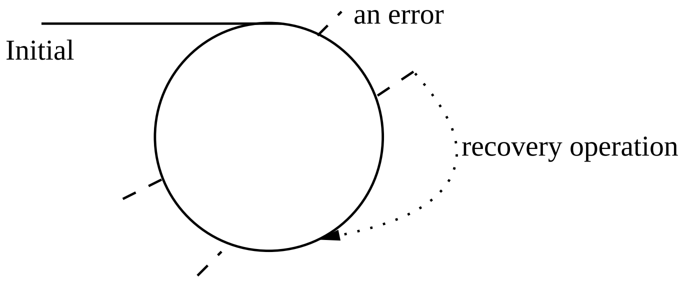

Fig. 3. Stabilization of the smart card personalization system.

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.

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 an 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 originating from physical damage or other reasons. This type of error is detected by the personalization stations. In $E1_a$ and $E1_b$ in Table 3 are examples of E1 error.
2. Type 2 error (E2) are errors originating from the personalization station when cards are personalized wrongly, which makes them unusable. This type of error is detected by a tester situated at the end of the personalization stations. $E2_a$ in Table 3 is an example of E2 error.

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.

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 retrieve a replacement card and place it in the right position. For these purpose we enrich our model with 'backward' and 'sweep' operations.

Table 3. The super single mode for 8 personalization stations with error. Only card values in station is shown

time	input	personalization stations								output
		0	1	2	3	4	5	6	7	(tester)
9										
10	□					4				
11	□	8								
12	□						5			
13	□		$E1_a$							
14	□							$E1_b$		
15	□			10?						
16	□								7?	
17	□				11?					
18										0
19	□					12?				1
20	□	16?								2
21	□						13?			$E2_a$
22	□		17?							4
23	□							14?		5
24	□			18?						$E1_b$
25	□								15?	7?
26	□				19?					
27										8

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 [2]. We assume that a backward move takes 1 time unit per step.

When the belt moves backward, the leftmost cards on the belt are also pushed back to the edge. For technical reasons explained in [2], 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 $\mathtt{d}_i$ ($\mathtt{0} \leq \mathtt{i} \leq \mathtt{D}$, $\mathtt{D} = \mathtt{M}$), 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 as:

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

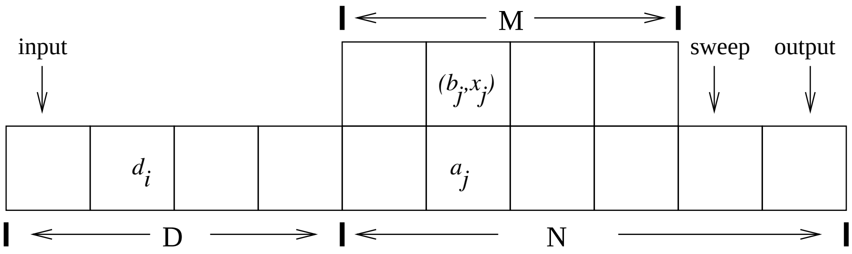

Fig. 4. Expanded model of the smart card personalization machine.

Table 4. Safety requirements for belt operations.

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

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. Complete SMV code for error recovery treatment is given in [8].

4.4 Results

For a single error scenario as defined above, there are `2.M` possible error configurations in one cycle of the super single mode. Using these error configurations as an initial state and the formula (3) we generated a recovery path that could stabilize the system back to the super single mode. Obviously, each path is different for different initial state, however, they share similar pattern. Thus we groups similar paths together and explain their property below.

1. When the error type is E1 and the faulty card was detected in the first half stations (b_i: $0 \leq i \leq \lfloor \frac{M}{2} \rfloor$), then the faulty card remain in the station until a free slot is available. And the personalization value remains unused until next. For example When the faulty card $E1_a$ in Table 3 was detected the personalization value (which is 9) was used in station 2
2. Using the same technique, for E1 errors in the second half stations (b_i: $\lfloor \frac{M}{2} \rfloor < i \leq M-1$) will not solve the problem, instead it will introduce another error.

The generated recovery path for this scenario is to skip the personalization value for now and let the error evolve to E2 error. The personalization value (6) of $E1_b$ in Table 3 was skipped and $E1_b$ will be again an error of type E2 at time 24.

3. The recovery path for E2 errors consists:
   - finding a station with a fresh card, this station should be in the first half. Otherwise an error like $E1_b$ will happen again. See also Example 1.
   - rolling the belt backward to this station,
   - personalizing the card with the personalizations value which is missing, and
   - dropping the card to the belt and forward it to the tester.

Example 1. In Table 5, at time 23 the 5^{th} card is found defective. At the same time station 6 starts with a fresh card. If a replacement card would be produced in this station, then personalization number 14 would be skipped. But this will 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 (station 2) that becomes available.

Table 5. Defective card treatment for error type 2.

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						
38	□							14		5

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 1.
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, based on the above observation, `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 present work shows error treatments for single error, we believe the same technique can be easily extended to multiple error treatment. In this way, using model checking, we go beyond scheduler synthesis and actually solve a control synthesis problem.

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

1. K. Åkesson and M. Fabian. Implementing supervisory control for chemical batch processes. In *International Conf. on Control Applications*, pages 1272–1277. IEEE Computer Society Press, 1999.
2. Sarah Albert. Cybernetix case study – informal description. Technical report, Cybernétix - LIF, 2002. Available through URL `http://ametist.cs.utwente.nl`.
3. B. A. Brandin. The real-time supervisory control of an experimental manufacturing cell. In *IEEE Transactions on Robotics and Automation*, volume 12, pages 1–13, 1996.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
5. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. KLuwer Academic, 1999.

6. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications ACM*, 17:643–644, 1974.
7. Ansgar Fehnker. Scheduling a steel plant with timed automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99),* Hong Kong, China, pages 280–287. IEEE Computer Society Press, 1999.
8. Biniam Gebremichael and Frits Vaandrager. Control synthesis for a smart card personalization system using symbolic model checking. Technical Report NIII-R0312, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, 2003.
9. J. Gunnarsson. *Symbolic Methods and Tools for Discrete Event Dynamic Systems.* PhD thesis, Linköping Studies in Science and Technology, 1997.
10. Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided Synthesis of Control Programs using UPPAAL. *Nordic Journal of Computing*, 8(1):43–64, 2001.
11. Tomas Krilavicius and Yaroslav Usenko. Smart card personalisation machine in UPPAAL and μCRL, 2003. In preparation.
12. Y. Li and W.M. Wonham. Control of vector discrete-event systems I - the base model. In *IEEE Trans. on Automatic Control*, volume 38, pages 1214–1227. IEEE Computer Society Press, 1993.
13. A. Mader. Deriving schedules for the cybernetix case study, 2003. Available through URL `http://ametist.cs.utwente.nl`.
14. K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.
15. Peter Niebert and Sergio Yovine. Computing optimal operation schemes for multi batch operation of chemical plants. In Nancy A. Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control, Third International Workshop, HSCC'00*, volume 1790 of *Lecture Notes in Computer Science*, pages 338–351. Springer, 2000.
16. P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.
17. Theo Ruys. Optimal Scheduling Using Branch and Bound with SPIN 4.0. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software – Proceedings of the 10th International SPIN Workshop (SPIN 2003)*, volume 2648 of *Lecture Notes in Computer Science*, pages 1–17, Portland, OR, USA, May 2003. Springer-Verlag, Berlin.
18. 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.
19. 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, 1991.
20. G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.
21. Gera Weiss. Modeling smart-card personalization machine with LSCs. Research report, Weizmann, 2003.
22. W.M. Wonham Y. Li. Control of vector discrete-event systems II - controller synthesis. In *IEEE Transactions on Autom. Control*, volume 39, pages 512–531, 1994.
23. Z. Zhang and W.M. Wonham. STCT: An efficient algorithm for supervisory control design. In *Symposium on Supervisory Control of Discrete Event Systems*, 2001.