Machine Checked Formal Proof of a Scheduling Protocol for Smartcard Personalization

Leonard Lensink, Sjaak Smetsers, and Marko van Eekelen

{L.Lensink, S.Smetsers, M.vanEekelen}@cs.ru.nl
Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands

Abstract. Using PVS (Prototype Verification System), we prove that an industry designed scheduler for a smartcard personalization machine is safe and optimal. This scheduler has previously been the subject of research in model checked scheduling synthesis and verification. These verification and synthesis efforts had only been done for a limited number of personalization stations. We have created an executable model and have proven the scheduling algorithm to be optimal and safe for any number of personalization stations. This result shows that theorem provers can be successfully used for industrial problems in cases where model checkers suffer from state explosion.

Keywords: verification, theorem proving, cyclic scheduling, simulation, PVS

1 Introduction

Formal methods provide the kind of rigor in software engineering that is needed to move the software development process to a level comparably to other engineering professions.

There are many kinds of formal methods that can be employed at different stages of the development process. In the specification phase, a model can be constructed using some kind of formal language. This model can be used as a starting point for model based testing. Model checking, which proves properties for the entire state space of a finite part of the formal model by means of an exhaustive test, can eliminate a lot of errors. Both model based testing and model checking can be performed automatically. Theorem proving can be used for full verification of models that can have an infinite number of states. However, employing theorem proving is considerably more costly than the earlier mentioned methods.

Formal verification of models is gaining ground within the industrial world. For instance, Cybernétix participated in the AMETIST project, in order to improve the quality of their systems. This project's aim was to develop modeling methodology supported by efficient computerized problem-solving tools for the modeling and analysis of complex, distributed, real-time systems. A personalization machine was one of the case studies supplied by Cybernétix. This machine consists of a conveyor belt with stations that personalize blank smartcards. The number of stations is variable.

The AMETIST participants modeled the machine in several model checking environments: Spin, Uppaal and SMV. However, within these systems, the models were checked and proven optimal and safe with respect to an ordering criterion for only a limited number of personalization stations. The most important reasons why it is interesting to look at the case study using other formal methods besides model checking are:

– In some production configurations the number of stations exceeds the amount of stations the model has been checked for. So there is not yet complete assurance that the scheduling algorithm is indeed safe and optimal for actually used configurations.
– Model checking is limited to a finite state space. Although there are methods allowing model checking to abstract away from the data or even to employ inductive reasoning on the model, so far no one has generalized to N stations. A stronger result would be to prove that for any number of stations, the scheduling algorithm is safe and optimal.
– Using a theorem prover to prove that a suitable invariant is correct usually gives more insight into why the machine satisfies its safety and optimality properties, instead of just checking them automatically.

In this paper we will present a formalized model of the machine in PVS (Prototype Verification System) [ORS92]. This is an environment for precise specification and verification of models. The specification language is based on simply typed higher order logic, but the type system has been extended with subtypes and dependent types. PVS also employs decision procedures to assist the user in a verification effort. These procedures take care of the bureaucracy associated with a formal proof and are usually able to discharge obvious proof obligations automatically. The specification language also allows for simulations and other means of animating the model if the model is composed out of an executable subset of the specification language.

We will come up with an invariant and use PVS to prove that this invariant holds for the model. This invariant is strong enough to prove all safety criteria and to prove that the algorithm guarantees optimal throughput for any number of personalization stations. We will also provide a simulation package. This makes it possible to verify that the model behaves as one would expect from a regular machine and which could form the basis of software that actually runs the machine.

In this article we present the smartcard personalization machine in section 2. The model of the machine is decribed in section 3 and we show by means of a simulation that this model is valid in section 4. Then, in section 5, the invariant is presented, followed by its proof in section 6. Safety and optimality are deduced from that invariant in section 6.1. A summary of related work by other people is given in section 7. An overview of future work can be found in section 8. All code and proofs referred to in this paper are available. [1]

[1] `http://www.cs.ru.nl/~leonard/papers/cybernetix/cybernetix.tar.gz`

2 Personalization machine

A smart card personalization machine takes blank smart cards as input and programs them with personalized data. Subsequently, the cards are printed and tested. Typically, a machine has a throughput of several thousands of cards per hour. The machine has a conveyor belt transporting the cards. There is an uploader station putting cards onto the belt and an unloader station taking them off again. Directly above the belt are posts that can manipulate the cards, either by lifting them off the belt, like personalization stations, or by processing the cards while they remain on the belt, like graphical treatment stations. An example configuration is given in figure 1.

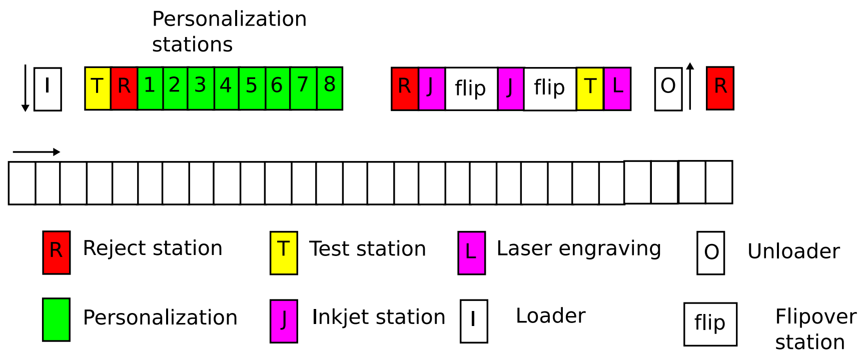


Fig. 1. Example of a standard configuration

There are different kinds of operations possible on the cards:

- Personalization stations program the chip on the card. These stations are able to detect if a card is defective. Cards need to be lifted into a personalization station by a lifting device.
- Graphical treatment stations are either laser engravers or inkjet stations. They can graphically personalize the cards. Graphical treatments happen while the card remains on the belt.
- Flipover stations can turn cards over to allow a graphical treatment of both sides of a card.
- Test stations determine whether the chip that is on the card functions properly.
- Rejection stations are used to extract cards that have been judged to be defective.

Due to the high number of cards that need to be personalized and the way the machine is structured, there are several requirements that need to be met by the smartcard personalization system:

– The output of the cards should happen in a predefined order, since further graphical treatment of the card may depend on the kind of personalization that has been received by the card. In the remainder of the paper we shall refer to this requirement as safety.
– The throughput of the machine should be optimal.
– The machine should allow for defective cards to be replaced.
– The system should be configurable and modular. The number of personalization and graphical treatment stations can vary according to the needs of the customers. Neither is the placement of the stations fixed. This means that the personalization stations can be spaced or appear interleaved with graphical treatment stations.

Cybernétix has developed and patented a scheduling protocol called "Super Single Mode". This particular scheduling protocol puts each time unit a new blank card on first position of the belt for N consecutive time units, where N is the number of personalization stations. After N time units, it leaves the first position of the belt empty for one time unit and then repeats itself by putting N new cards on the belt followed by leaving one slot empty.

3 PVS Model of the personalization machine

In the previous section, we have given a general description of the personalization machine. In this section we will discuss the model we have developed.

The personalization machine is modeled as a conveyor belt that transports cards underneath a set of M personalization stations. Each of these stations can pick up and drop cards onto the conveyor belt. The belt is synchronized with the personalization stations in order to enable picking up and dropping the cards.

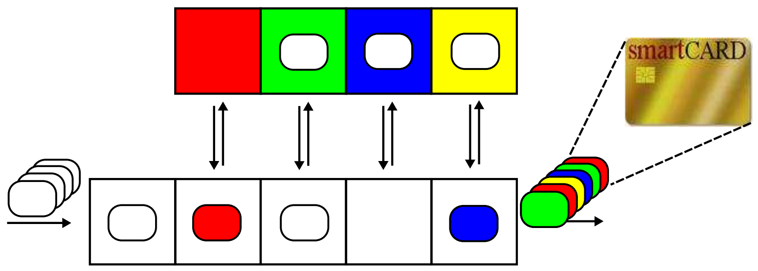


Fig. 2. A simplified machine with 4 stations

Since we are interested in the scheduling mechanism, the model that has been constructed can ignore several aspects of the machine, similarly to other studies [GV04,Ruy03,HKW05].

– For the scheduling algorithm it is not relevant how the cards end up on the belt or how they are taken off. This means that the loader and unloader can be safely omitted from the model.

– We assume that no cards are defective. This means that there is no need to model neither the testing stations nor the stations that take rejected cards off the belt. Although this reduces the interest of the example, only the study by Gebremichael [GV04] addressed the failed cards by creating a special "faulty" card mode. This can be added to the generalized model without too much effort in a later stage.

– The graphical treatment and flipover stations have also been omitted. These stations do not take cards off the belt, so they can not interfere with the ordering on the belt. Also, the processing time is magnitudes smaller than the processing time of the personalization stations. They have a negligible impact on the throughput of the system.

– The loading and offloading time of the personalization stations is also much smaller than the personalization time and not included into the model.

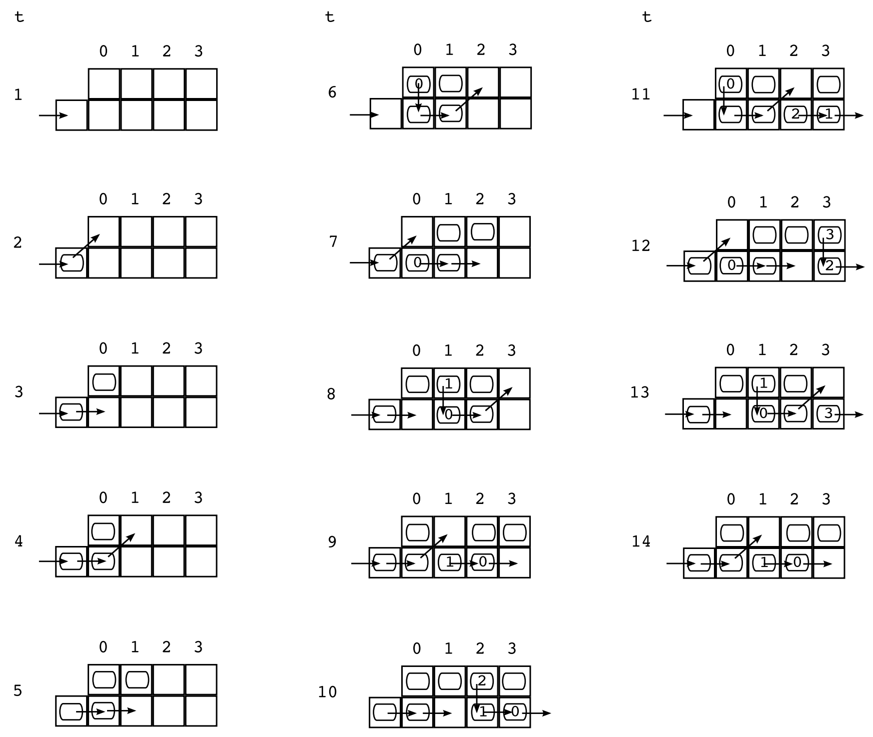


Fig. 3. Personalization run in super single mode

When the machine is started, the belt and all the personalization stations are empty. In figure 3 we show the transition of a four station personalization machine through time. At each transition, the belt is moved one slot and subsequently the cards are dropped or lifted from the slots when needed. The arrows indicate the move a card is about to make. The numbers above the stations indicate the kind of personalization produced by that station and can also be found on the card after a station has finished personalizing and has dropped the card back onto the conveyor belt.

At first, when the time that has passed is smaller than 9, the system is in an initial state where all the stations fill up with cards being processed. At `t=9`, the system starts a cycle that lasts for five transitions. As one can see in figure 3, the state at `t=14` is the same as at `t=9`. The state of the machine as depicted in figure 2 can be found in the table at `t=12`.

Our aim in constructing a PVS model is to verify that the scheduling algorithm satisfies the following criteria:

– The personalized cards should leave the machine in the order of the occurrence of the personalization stations. Cards personalized by station 0 should appear at the last slot on the belt before the card personalized by station 1. No other sorting mechanism may exist in the system.
– The throughput of the machine should be optimal.

3.1 The belt

The model encodes the conveyor belt using an algebraic data type. A slot on the belt can either contain no card: `empty`, contain a smartcard that has yet to be personalized: `new_card`, or contain a personalized card: `personalization`. The personalization is modeled as a natural number that corresponds to the relative position of the personalization station with respect to the conveyor belt. This means that cards leaving the left most station get `0`, and the rightmost `M`.

In PVS, algebraic data types are specified by providing the *constructors* as well as *recognizers* and *accessors*. The constructors `empty`, `new_card` and `personalization` are used to build the objects of that data type. The recognizers (`empty?`,`new_card?` and `personalization?`) are used to determine of which kind an expression of the `slot` type is and the accessor `number` can be used to extract the `personalization_nr`, in case of a personalization.

```
slot : DATATYPE
BEGIN
empty : empty?
new_card : new_card?
personalization (number : personalization_nr) : personalization?
END slot
```

The conveyor belt is modeled as an array of $1 + M$ of these slots. Each slot is indexed by a natural number from `0` up to `M`. In PVS, these restrictions on values which can be held by an object can be expressed elegantly using *dependent* types: types dependent on values. For example, the (finite) subset $\{0, \ldots, M\}$ of the natural numbers can be described as `below(n:nat) : TYPE = { m : nat | m < n }`. In this case, the predicate on the natural numbers is `below(1+M)`.

```
beltposition : TYPE = below(1+M)
```

3.2 The stations

The relevant information to model a personalization station is whether a station is programming a card and if so, how far the personalization process has progressed. A timer is used to model this. The value `0` is assigned to a station to indicate that a station is `empty` and not working on a card. Once a station starts personalizing, the value is increased to `1` and incremented each time slot until it reaches the time needed to complete the personalization process. At that time, the machine will start looking whether it can drop the card or not. Theoretically, the machine can keep incrementing the timer as long as the card has not been dropped. Therefore, we model the timer by a natural number.

```
timer : TYPE = nat
```

Since we have one less personalization station than there are slots on the belt, the stations are modeled as an array of M of these timers.

```
stationposition : TYPE = below(M)
```

3.3 The machine

The entire machine is rather straightforward. The machine is viewed as an array of M stations combined with an array of $1+M$ belt-slots. A global timer is used to synchronize actions on the belt and in the stations. In PVS this is modeled using a record type:

```
machine_state : TYPE =
  [# stations : [stationposition → timer],
     belt : [beltposition → slot],
     global_timer : global_timer
   #]
```

In figure 4 the machine as earlier depicted in figure 2 is shown as a representation of the PVS model.

The behavior of the machine is described by a function `f_next`. This function transforms a machine state into the next machine state by operating the belt slots and stations for each position and by increasing the global timer. The next state of a station and belt at a certain position is determined by the content of the previous belt slot or the previous station.

- In the case of a station, the next state can only be determined by the content of the belt that is situated to the left and below the station. In the model they are indexed by the same position number.

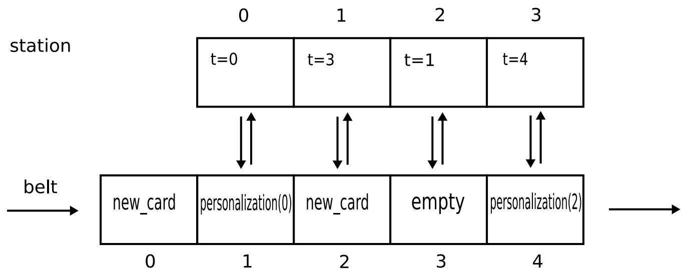


Fig. 4. Model of the simplified machine from figure 2

– In the case of the belt, the next state at a certain position is determined by the content of either the station directly above the belt or the previous belt position. Both are indexed by the position minus one.

The `f_next` function constructs the next state out of the current state by creating a new record of type `machine_state`:

```
f_next(ps:machine_state) : machine_state =
  (# stations := f_operate_station(ps)
   , belt := f_operate_belt(ps)
   , global_timer := global_timer(ps)+1
   #)
```

The behavior of the machine is best described by discerning three different situations:

1. *We have an empty station and a new card is available on the previous slot on the belt.* In this case, we move the card from the belt into the station and start personalizing. As a consequence, the belt position becomes empty and the station's timer is started.
2. *The timer in the station indicates that the card has been personalized and there is an empty spot on the belt.* This means the personalized card, which is designated by its position, can be dropped onto the belt, leaving an empty station. At the same time the timer is reset.
3. *If none of the above applies* the contents of the belt are just shifted one position. *If the station at the position is personalizing* we adjust the timer by one tick to denote the progress of time.

The function operating on each station checks whether the timer of the station needs to be started, reset or increased, depending on whether it is done personalizing cards or ready to take in a new card:

```
f_operate_station(ps:machine_state)(spos:stationposition) : timer =
 LET station = station(ps)(spos), belt = belt(ps)(spos) IN
 IF empty?(station) ∧ new_card?(belt)
 THEN start_timer
 ELSIF done?(station) ∧ empty?(belt)
 THEN reset_timer
 ELSIF ¬empty?(station)
 THEN increase_timer(station)
 ELSE wait(station)
 ENDIF
```

The function that operates the belt reacts to basically the same conditions as the previous function with exception of the first belt position. There the cards must be scheduled according to the scheduling algorithm:

```
 f_operate_belt(ps:machine_state)(bpos:beltposition) : slot =
  IF bpos=0
  THEN schedule(global_timer(ps))
  ELSE
   LET station = station(ps)(bpos-1), belt = belt(ps)(bpos-1) IN
   IF empty?(station) ∧ new_card?(belt)
   THEN lift
   ELSIF done?(station) ∧ empty?(belt)
   THEN drop(bpos)
   ELSE move_belt(belt)
   ENDIF
  ENDIF
```

The behavior of the system strongly depends on the time a personalization station needs to finish. If the personalization time exceeds the number of personalization stations, the safety property will not be satisfied, because it will mean that a blank card will reach the end of the conveyor belt before one of the stations will be able to pick it up. If the personalization time is smaller than M, there will not be an empty spot available to drop the card. This spot will only arrive after M time units, so it makes sense to have the personalization end at that time.

```
 done?(t:timer) : bool = t=M
```

3.4 The scheduler

The scheduler is a process that puts the cards onto the first spot of the conveyor belt in a cyclic fashion. It places M new cards on the belt followed by an empty spot. In order to keep track of when an empty space should be left on the belt, the global timer is used:

```
schedule(global_timer:global_timer) : slot =
 IF mod(global_timer,1+M) = 0
 THEN empty
 ELSE new_card
 ENDIF
```

4 Validating the model

In section 3, we developed a model of the personalization machine. When modeling a system, the key question is whether it faithfully represents the original machine. In order to show this is indeed the case we need to be able to execute our model and make a visual representation that mimics the behavior expected from a personalization machine. This approach provides us with several benefits:

- To prove the safety property of the machine an invariant is needed. Visualizing the behavior makes is easier to determine this invariant.
- Secondly, if we have an appropriate API to drive the belt and sensors, the executable model means that we can generate code to run the machine. No manual translation from model to code is necessary. This eliminates a possible source of errors.
- Finally, visualizing the behavior of the model allows us to verify that the model behaves as expected.

PVS allows for animation of its specifications by means of a ground evaluator [vHPPR98]. The evaluator extracts executable Common Lisp code from the PVS functional specifications. Semantic attachments enable a safe connection of user defined Lisp functions to uninterpreted PVS functions. A library, PVSio [Muñ03], extends the ground evaluator with a library of predefined functions to handle all kinds of imperative languages features.

Since we have written the model in PVS, using only functional specifications, it is directly executable by PVS' ground evaluator. On top of the executable model it is possible to add IO as a side-effect of the original statements. Functions that produce side-effects must be modeled as Boolean functions that always return true. By conjoining those functions with the original model they will be executed alongside the executable model. We define a simulation function that takes as arguments how many times the transition should take place and the starting state. As a side effect, the state of the machine is printed to the standard output so we can observe the machine as time progresses.

```
f_step(ps:machine_state)(p:nat) : RECURSIVE void =
   print_state(ps) ∧
   (
   IF (p=0)
   THEN println("End of simulation")
   ELSE f_step(f_next(ps))(p-1)
   ENDIF
   )
 MEASURE pn
```

The function `print_state(ps0)` prints the state variables to the standard output.

Although no machine experts were involved in validating this particular model, the models from the original AMETIST project were. The PVS model is close enough to these models to validate it against its expected behavior. We have simulated behavior for machines of several sizes and as an example show the validation of a conveyor belt with four personalization stations. What should be expected is earlier depicted in figure 3. A `#` denotes a new card, a `*` denotes a station that is personalizing, ˆ an empty station, `!` shows a station that is done personalizing, while the natural numbers stand for personalized cards. In figure 5 we show the output generated by a simulation run of the model for a four station machine.

```
  <PVSio> simulation(14);

 1    ^ ^ ^ ^                    8    * ! * ^
   __________                       _#___0_#__

 2    ^ ^ ^ ^                    9    * ^ * *
   _#________                       _#_#_1_0__

 3    * ^ ^ ^                   10    * * ! *
   _#________                       _#_#___1_0

 4    * ^ ^ ^                   11    ! * ^ *
   _#_#______                       ___#_#_2_1

 5    * * ^ ^                   12    ^ * * !
   _#_#______                       _#_0_#___2

 6    ! * ^ ^                   13    * ! * ^
   ___#_#____                       _#___0_#_3

 7    ^ * * ^                   14    * ^ * *
   _#_0_#____                       _#_#_1_0__
```

Fig. 5. A simulation run in PVSio

A comparison of figure 5 with figure 3 shows that the simulation behaves as expected.

5 The complete state invariant

In section 4 we have shown by means of a simulation that the model behaves as expected for four stations. The next step is to prove that the model satisfies the safety and optimality requirements:

- Concerning the safety property: The machine must maintain the order of the personalization stations in its generated output order. This can be split up in two requirements.
  - First, only personalized cards or empty spaces should be present at the last slot of the belt.
  - Secondly, once a personalized card n, where $0 \leq n < M$, is present at the final position on the belt, the next card has to be personalization $mod(n+1, M)$ or a sequence of empty slots until the next card is personalization $mod(n+1, M)$.

– Concerning the optimality property: The machine must personalize as many cards as possible per time unit. The optimum is reached if all personalization stations are occupied and personalizing all of the time. This means that once the cyclic phase of the machine is entered, two properties should hold:
  - If a station is empty, then it must immediately be able to load a new card and start personalizing.
  - If a station is done personalizing, an empty space should immediately be available to drop the card.

We can formulate the safety property slightly more specific, because we know that only one empty spot is scheduled each cycle. This means that there can be only one empty spot in the output position once the cyclic phase of the machine has been reached. As a consequence, we can conclude that the order in which the personalized cards leave the machine must be linearly related to the value of the `global_timer`. We have established that the relation between the value of the `global_timer` and the value of the personalized card, `number(belt(ps)(M))`, however, we do not know yet at what time exactly `mod(global_timer(ps),1+M)` will be equal to personalized card `0`. There might be a phase transposition. We call this `c`.

Assuming we have M personalization stations the first property can be specified formally as:

```
  empty?(belt(ps)(M))
∨ (personalization?(belt(ps)(M)) ∧
   ∃ c: mod(global_timer(ps)+c,1+M) = number(belt(ps)(M)))
```

The second property can be formally specified as:

```
∀ pos: ∃ ps': global_timer(ps') = global_timer(ps)+1 ∧
(empty?(station(ps)(pos)) ⇒ start?(station(ps')(pos))) ∧
(done?(station(ps)(pos)) ⇒ empty?(station(ps')(pos)))
```

Trying to prove these two properties directly turns out to be futile. In order to prove them we need to come up with an invariant that is stronger than the safety and optimality properties. More particularly, in this invariant must be expressed that whenever a station has finished personalizing, an empty spot will be available to deposit the personalized card.

We assume the machine starts with an empty belt and all stations empty. After an initialization phase, the machine will end up in a cyclic state until the machine is shut down. In the initialization phase, the stations and belt positions remain empty, until an empty card reaches them.

The graphical representation of the state of the personalization machine, devised to validate the working of the system can also be put to good use in deriving the invariant needed to prove the relevant properties.

In figure 6, the first observation we make is that the cyclic phase propagates through the positions at the rate of one position every two time units. After two time units the first position satisfies the stable (cyclic) invariant, while the rest of the belt still is in its initial state. After four time units, the first two positions

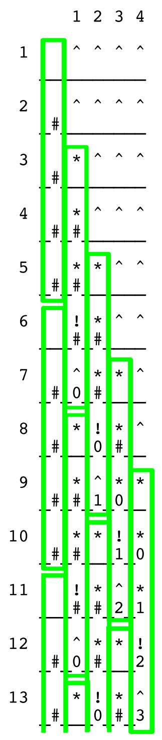


Fig. 6. Cyclic invariant propagation

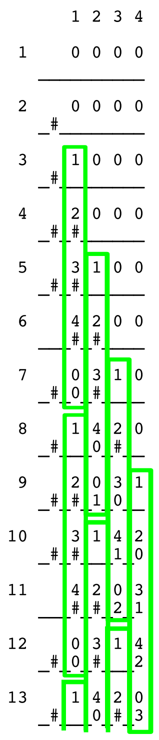


Fig. 7. State of the stations

satisfy the invariant, while the remaining part of the belt and stations are still in their initial state, and so on:

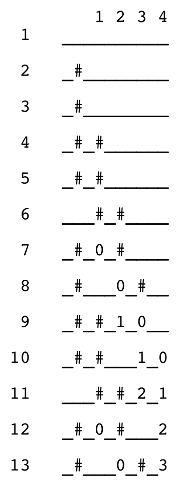
```
 p_invariant(ps:machine_state) : bool =
  ∀ bpos : IF 2*bpos+1 ≥ global_timer(ps)
           THEN p_init(ps)(bpos)
           ELSE p_stable(ps)(bpos)
           ENDIF
```

The initial invariant is simply that the timer of the station at position *pos* is 0 and consequently the station is empty, as well as the corresponding belt position.

```
p_init(ps:machine_state)(bpos:beltposition) : bool =
 (bpos ≤ M-1 ⇒ station(ps)(bpos) = 0) ∧ empty?(belt(ps)(bpos))
```

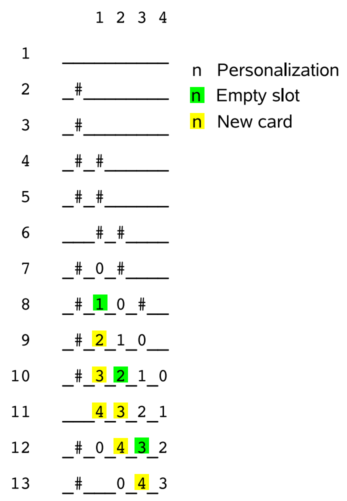
Observations on the stations of the personalization machine allow us to conclude that the timer of a station is related to the value of the global timer. As seen in figure 7, the value of a station neatly increases in time with a phase difference according to its position: `station(bpos) = mod(global_timer - 2*(bpos+1),1+M)`

The relationship between the global timer and the contents of the belt at a certain position are slightly more complex. In order to clarify that relationship, the state of the stations is removed from the representation in figure 8.



```
      1 2 3 4
1   __________
2   _#________
3   _#________
4   _#_#______
5   _#_#______
6   ___#_#____
7   _#_0_#____
8   _#___0_#__
9   _#_#_1_0__
10  _#_#___1_0
11  ___#_#_2_1
12  _#_0_#___2
13  _#___0_#_3
```

Fig. 8. State of the belt



```
      1 2 3 4
1   __________      n Personalization
2   _#________      n Empty slot
3   _#________      n New card
4   _#_#______
5   _#_#______
6   ___#_#____
7   _#_0_#____
8   _#_1_0_#__
9   _#_2_1_0__
10  _#_3_2_1_0
11  ___4_3_2_1
12  _#_0_4_3_2
13  _#___0_4_3
```

Fig. 9. State of the belt with selected numerical representations

We replace some of the symbols we have used with a numerical representation. From this representation as in figure 9 we can derive the following property for the content of the belt:

```
belt = mod(global_timer(ps)-bpos-1,1+M) ∧
IF belt = bpos THEN empty
ELSIF belt > bpos THEN new_card
ELSE personalization(number(belt))
ENDIF
```

Combining and rewriting the above results we obtain an invariant for the entire system:

```
p_stable(ps:machine_state)(pos:beltposition) : bool =
(pos ≤ M-1 ⇒ mod(global_timer(ps)-2*(pos+1),1+M) = station(ps)(pos))
∧
LET timer = mod(global_timer(ps)-2*pos-1,1+M), belt = belt(ps)(pos) IN
IF timer = 0
THEN  empty?(belt)
ELSIF timer < 1+M-pos
THEN new_card?(belt)
ELSE personalization?(belt) ∧ number(belt) = timer-1-M+pos
ENDIF
```

Since it contains complete information of the state of the machine at any given time, it should be possible to prove that this invariant (if it is correct) holds. We call this the *complete state invariant*. From this invariant, we can then directly deduce the properties we want to prove.

6 Proof of the complete state invariant

After specifying the invariant in PVS, we will now prove that the invariant holds in the initial state and does not change with each consecutive state change. We define the following theorem within PVS:

```
invariant: THEOREM
 p_invariant(ps_init) ∧  (p_invariant(ps) ⇒ p_invariant(f_next(ps)))
```

Proving the invariant to hold is done by case distinctions on the invariant, as well as case distinctions on the functions `f_operate_belt` and `f_operate_station`. These distinctions then invariably lead to some equation that can be proven correct using modulo arithmetic or to a contradiction within the assumptions. In the standard library of PVS, there are a number of lemmas that are sufficient to discharge all of the modular proof obligations. To provide better understanding, we describe a part of the proof in detail: We want to prove that the transition in the first part of the `f_operate_station` function does not invalidate the invariant. The relevant part of the function is:

```
[..]
  IF empty?(station(ps)(pos)) ∧ new_card?(belt(ps)(pos))
  THEN start_timer
[..]
```

Where `start_timer` returns the timer value of `1`.

It has to be shown that the invariant still holds if `empty?(station(ps)(pos))` and `new_card?(belt(ps)(pos))` then `station(f_next(ps))(pos) = 1` is added to the assumptions. This simplifies the invariant to two items that have to be proven:

– First, `2*pos+1 < 1+global_timer(ps)`. This can be derived from the fact that the `p_init(ps)` part of the invariant has to be false. The value of `station(ps)(pos)` is one, while the invariant states that it is zero when `2*pos+1 >= 1+global_timer(ps)`.
– Secondly, filling out the invariant further with the knowledge that the timer of the station at position `pos` is one and assuming that we can prove the first of our proof obligations the part of the invariant that remains is:

  ```
  mod(1+global_timer(ps)-2*pos,1+M) = 1
  ```

Because we know that at time `global_timer(ps)` we had a new card at the previous position, the invariant adds to the assumptions:

```
mod(global_timer(ps)-2*pos,1+M) < 1+M-pos
```

From this assumption, using modulo arithmetic it is deducible that:

`global_timer(ps)` $\geq$ `2*pos`

There are two possible cases left:

- Either `global_timer = 2*pos`. Then, again using modulo arithmetic, it is easy to prove that `mod(1+global_timer(ps)-2*pos,1+M) = 1`.
- Otherwise, `2*pos > global_timer(ps)`. Then we know that the stable part of the invariant holds at `global_timer(ps)`.
  This means: `mod(global_timer-2*pos,1+M) = 0`. This can be proven using modulo arithmetic.

The other situations where personalized cards are dropped in empty slots or the card on the belt is just moved to the right and the timer in the station is optionally increased are slightly more complicated, but revolve around a number of case distinctions as well. The total proof, which is surely not optimized, needs about 250 proof commands in PVS to be performed completely. Creating the model, deriving the invariant and proving the invariant to hold, took about a month for a PhD student, relatively inexperienced with PVS.

6.1 Safety and optimality

Now that we have established that the invariant holds at all times, we will prove that the safety and optimality properties follow directly from the invariant:

- The safety property meant that the personalized cards leave the personalization part of the machine in order of the kind of personalization they have received. Once the invariant has been proven to hold, it follows directly that at the end of the conveyor belt (at position `M`), the following holds:

  $$\begin{array}{l} \quad \texttt{empty?(belt(ps)(M))} \\ \lor (\texttt{personalization?(belt(ps)(M))} \land \\ \quad\; \texttt{mod(global\_timer(ps),1+M)} = \texttt{number(belt(ps)(M)))} \end{array}$$

  Since `global_timer(ps)` is ordered, `mod(global_timer(ps)),1+M)` is ordered as well.
- The optimality property implied that the scheduling protocol needs to have the highest throughput per cycle. This derives immediately from the fact that if we have `1+M` consecutive cards, the machine will not be able to personalize all the cards. This will violate the safety requirements. Therefore, the highest throughput per cycle is reached by leaving only one empty slot after M consecutive cards.

7 Related work

The Cybernétix case study has been the subject of several research papers. Kugler and Weiss wrote an article about how to interactively derive scheduling algorithms for production lines using Live Sequence Charts [HKW05]. In it, they use a graphical representation to analyze a production line systematically. However, no properties for that production line are proved. In [Mad04] Mader compares two different scheduling algorithms using model checking, for four and eight personalization stations, but the model checking was limited to a maximum of respectively 16 and 40 personalized cards. In contrast to the other studies, Mader does include the graphical treatment in her model. Ruys uses new features of SPIN 4.0 to derive an optimal schedule for four stations and at most five cards [Ruy03]. Nieberg proves in [Nie04] with a mathematical argument that the Super Single Mode is optimal, but does not provide a formal proof that the protocol is safe with respect to the ordering of the cards. Also using model checking, Gebremichael [GV04] is able to derive the Super Single Mode as an optimal schedule for five personalization stations and any number of cards. Gebremichael also extends his model to deal with a possible defective card. None of the studies concerning the smartcard personalization machine combine the rigor of machine checked proof and simulation with a general proof of optimality and safety. In PVS work has been done to integrate model checking and theorem proving for models that have a finite number of states as described in [RSS95]. However, these models must conform to some syntactic restrictions that complicate actually using the model checking part of PVS in practice. Work on verifying algorithms and code generation from PVS has been done by Jacobs, Wichers Schreur and Smetsers in [JSS07], where executable parts PVS specifications are translated into the functional programming language Clean.

8 Future work

The ad hoc nature of the derivation of the invariant needed for the proof of the properties, suggests a natural direction for future work. More case studies can hopefully give us ideas how to derive invariants more methodically. We have only focused on the scheduling mechanism on a rather abstract level until now. If code that drives the machine is to be generated, more detail will have to be added to the specification. An open question is whether the proof will have to be substantially altered when this is attempted. Another subject of research concerns devising methods to incorporate the context in which the generated code has to be run into the theorem prover itself in a methodical and easy to use fashion.

9 Conclusion

We addressed the Cybernétix smartcard personalization machine as an example of an industry supplied case study for the application of formal methods. We constructed an executable model in the specification language PVS. Since the model is executable it was straightforward to visualize the behavior of the model and construct a simulator that was used to establish that the model that had been created adequately represented the machine. In future work it is possible to use the verified scheduling algorithm to control the machine itself, eliminating any errors that might arise from manually translating the model into code.

Model checking techniques already proved optimality and safety of this machine for a limited number of stations. In typical applications of this machine, the number of stations will be much larger than the amount for which was model checked. This means that no guarantee can be given that the properties will hold generally. By using a theorem prover we have established that the safety and optimality of the scheduling algorithm is guaranteed for any number of personalization stations.

References

[GV04] B. Gebremichael and F.W. Vaandrager. Control synthesis for a smart card personalization system using symbolic model checking. In K.G. Larsen and P. Niebert, editors, *Proceedings First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, volume 2791 of *LNCS*, pages 189 – 203. Springer Verlag, 2004.

[HKW05] D. Harel, H. Kugler, and G. Weiss. Some methodological observations resulting from experience using lscs and the play-in/play-out approach. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 26–42, Berlin / Heidelberg, august 2005. Springer.

[JSS07] Bart Jacobs, Sjaak Smetsers, and Ronny Wichers Schreur. Code-carrying theories. *Formal Aspects of Computing*, 19(2):191–203, June 2007.

[Mad04] A. H. Mader. Deriving schedules for a smart card personalisation system. Technical Report TR-CTIT-04-05, University of Twente, Enschede, January 2004.

[Muñ03] C. Muñoz. Rapid prototyping in PVS. Report NIA Report No. 2003-03, NASA/CR-2003-212418, NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, May 2003.

[Nie04] T. Nieberg. On cyclic plans for scheduling a smart card personalisation system. Technical Report TR-CTIT-04-01, Centre for Telematics and Information Technology, University of Twente, Enschede, January 2004.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, jun 1995. Springer-Verlag.

[Ruy03] T. C. Ruys. Optimal scheduling using branch and bound with spin 4.0. In *Model Checking Software: 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 1–17. Springer Berlin / Heidelberg, May 2003.

[vHPPR98] F. von Henke, S. Pfab, H. Pfeifer, and H. Rueß. Case studies in meta-level theorem proving. In *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 461–478, Berlin / Heidelberg, Sept. 1998. Springer.