Verification and Optimization of a PLC Control Schedule

Ed Brinksma[1] and Angelika Mader[2] *

[1]Faculty of Computer Science, University of Twente
[2]Computer Science Department, University of Nijmegen

Abstract. We report on the use of the SPIN model checker for both the verification of a process control program and the derivation of optimal control schedules. This work was carried out as part of a case study for the EC VHS project (Verification of Hybrid Systems), in which the program for a Programmable Logic Controller (PLC) of an experimental chemical plant had to be designed and verified. The intention of our approach was to see how much could be achieved here using the standard model checking environment of SPIN/Promela. As the symbolic calculations of real-time model checkers can be quite expensive it is interesting to try and exploit the efficiency of established non-real-time model checkers like SPIN in those cases where promising work-arounds seem to exist. In our case we handled the relevant real-time properties of the PLC controller using a time-abstraction technique; for the scheduling we implemented in Promela a so-called *variable time advance procedure*. For this case study these techniques proved sufficient to verify the design of the controller and derive (time-)optimal schedules with reasonable time and space requirements.

1 Introduction

Nowadays, the verification of hybrid systems is a popular topic in the formal methods community. The presence of both discrete and continuous phenomena in such systems poses an inspiring challenge for our specification and modelling techniques, as well as for our analytic capacities. This has led to the development of new, expressive models, such as timed and hybrid automata [3, 16], and new verification methods, most notably model checking techniques involving a symbolic treatment of real-time (and hybrid) aspects [10, 17, 6].

An important example of hybrid (embedded) systems are process control programs, which involve the digital control of processing plants, e.g. chemical plants. A class of process controllers that are of considerable practical importance are those that are implemented using Programmable Logic Computers or PLCs. Unfortunately, both PLCs and their associated programming languages have

* supported by an NWO postdoc grant and the EC LTR project VHS (project nr. 26270)

no well-defined formal models, c.q. semantics, which complicates the design of reliable controllers and their analysis.

To assess the capacity of state-of-the-art formal methods and tools for the analysis of hybrid systems, the EC research project VHS (Verification of Hybrid Systems) has defined a number of case studies. One of these studies concerns the design and verification of a PLC program for an experimental chemical plant.

In this paper we report on the use of the SPIN model checker for both the verification of a process control program for the given plant and the derivation of optimal control schedules. It is a companion paper to [12], which concentrates on the correct design of the process controller. The intention of our approach was to see how much could be achieved here using the standard model checking environment of SPIN/Promela [7]. As the symbolic calculations of real-time model checkers can be quite expensive it is interesting to try and exploit the efficiency of established non-real-time model checkers like SPIN in those cases where promising work-arounds seem to exist. In our case we handled the relevant real-time properties of the PLC controller using a time-abstraction technique; for the scheduling we implemented in Promela a so-called *variable time advance procedure* [15]. For this case study these techniques proved sufficient to verify the design of the controller and derive (time-)optimal schedules with very reasonable time and space requirements.

The rest of this paper is organised as follows: section 2 gives a description of the batch plant, the nature of PLCs and a description of the control program that was systematically designed in [12]. Section 3 describes the Promela models for the plant and the control process, and their use for the formal verification and optimization. Section 4 contains the conclusions.

2 Description of the system

The system of the case study is basically an embedded system, consisting of a batch plant and a Programmable Logic Controller (PLC), both of which are described in more detail below. The original goal of the case study was to write a control program such that the batch plant and the PLC with its control program together behave as intended. The intended behaviour is that, first, new batches can always be produced, and second, in the second place, that the control schedule is time optimal, i.e. the average time to produce a batch is minimal.

2.1 Description of the batch plant

The batch plant (see Figure 1) of the case study is an experimental chemical process plant, originally designed for student exercises. We describe its main features below; a more detailed account can be found in [9]. The plant "produces" batches of diluted salt solution from concentrated salt solution (in container B1) and water (in container B2). These ingredients are mixed in container B3 to obtain the diluted solution, which is subsequently transported to container B4 and then further on to B5. In container B5 an evaporation process is started. The evaporated water goes via a condenser to container B6, where it is cooled and pumped back to B2. The remaining hot, concentrated salt solution in B5 is transported to B7, cooled down and then pumped back to B1.

The controlled batch plant is clearly a hybrid system. The discrete element is provided by the control program and the (abstract) states of the valves, mixer, heater and coolers (open/closed, on/off). Continuous aspects are tank filling levels, temperatures, and time. The latter can be dissected into real-time phenomena of the plant on the one hand, such as tank filling, evaporation, mixing, heating and cooling times, and the program execution and reaction times (PLC scan cycle time), on the other. The controller of the batch plant is a nice example of an *embedded system*: the controlling, digital device is part of a larger physical system with a particular functionality.

For the case study we decided to fix the size of a batch: the material is either 4.2l salt solution with a concentration of 5g/l and 2.8l water or, if mixed, 7l salt solution of 3g/l concentration. With these batch sizes containers B1, B2, B4, B6 and B7 are capable of two "units" of material, B3 and B5 only one "unit" of material. The plant description in [9] contains also durations for the transport steps from one tank to another. In our (timed) plant model we used these durations as our basis, although the actual durations might possibly be different.

B1–B3	B2–B3	B3–B4	B4–B5	heat B5	B5–B7	cool B6	cool B7	B6–B2	B7–B1
320	240	70	350	1100	280	300	600	240	220

Table 1. Duration of plant processes in seconds

2.2 Programmable Logic Controllers

PLCs are special purpose computers designed for control tasks. Their area of application is enormous. Here, we briefly emphasize the main characteristics of PLCs in comparison to "usual" computers.

The most significant difference is that a program on a PLC runs in a permanent loop, the so called *scan cycle*. In a scan cycle the program in the PLC is executed once, where the program execution may depend on varible values stored in the memory. The length of a scan cycle is in the range of milliseconds, depending on the length of the program. Furthermore, in each scan cycle there is a data exchange with the environment: a PLC has *input points* connected via an interface with a dedicated *input area* of its memory, and the *output area* of the memory is connected via an interface with the *output points* of the PLC. On the input points the PLC receives data from sensors, on the output points the PLC

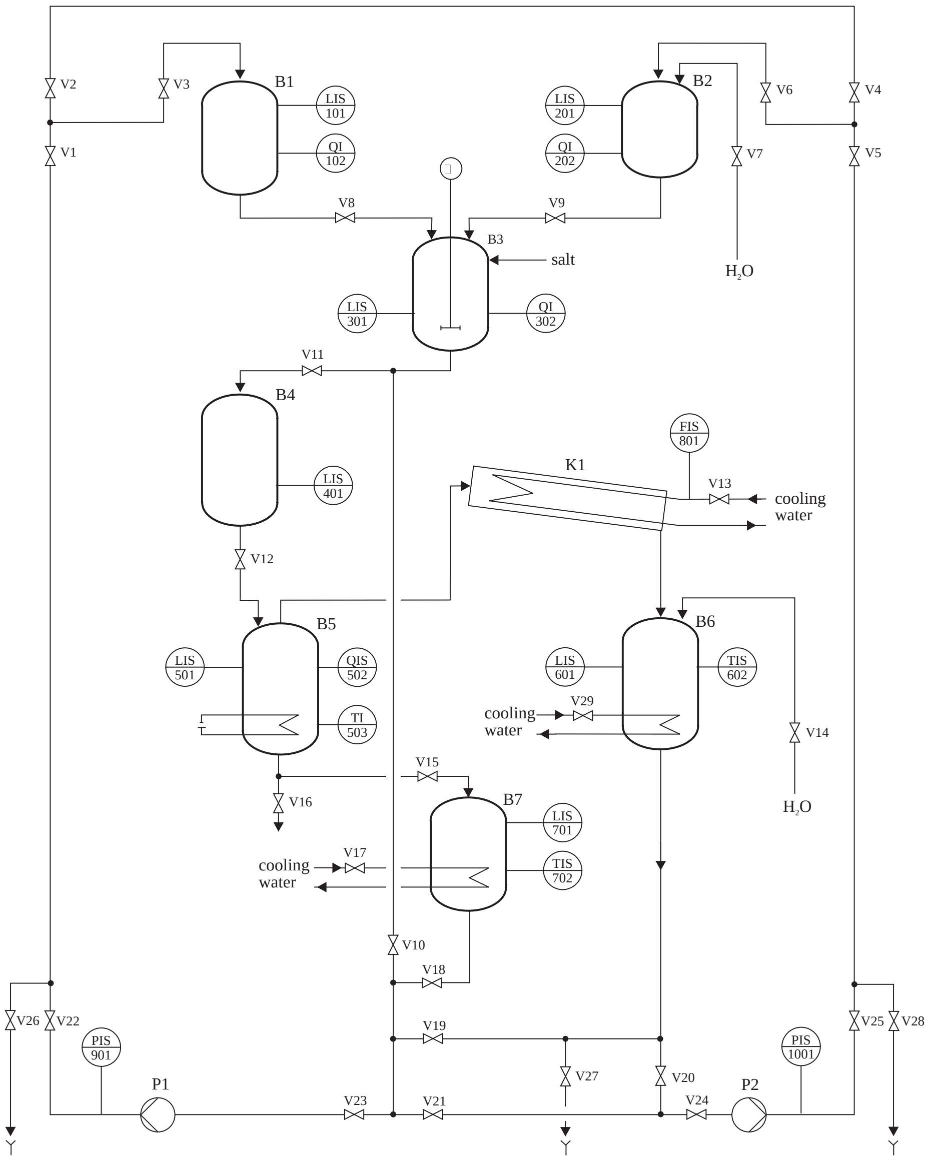

Fig. 1. The P/I-Diagram of the Batch Plant

sends data to actuators. Finally, there are some activities of the operating system (self checks, watchdogs etc.) that take place in a scan cycle. The operating system itself is small and stable, which is prerequisite for reliable real-time control. PLC programs are developed and compiled on PCs in special programming environments and can be loaded to the PLC. There are different programming languages collected in a standard [8]. In our application we used Sequential Function Charts (SFC), a graphical language that is related to Petri-Nets, and the program executed in each scan cycle depends on the places that are active at the moment. In this sense SFC provides a meta-structure and the actual instructions of our application are written in Instruction List, an assembly-like language. In these languages it is possible to make use of *timers* which is also a difference to the programming languages we usually deal with.

The scan cycle mechanism makes PLCs suitable for control of continuous processes (tight loop control). However, it has to be guaranteed that the scan cycle length is always below the minimal reaction time that is required by the plant to control the entire system. In this case study the scan cycle time is a few orders of magnitude smaller than what the reaction time has to be. The execution time of a scan cycle is in the range of a few milliseconds. For some applications the timing behaviour in this range is relevant, e.g. for machine control. For our chemical plant it is not relevant: it does not really matter whether a valve closes 10 ms earlier or later. This property is relevant when modelling the whole system. Here, we can model the PLC as if executing “time-continuously”, i.e. a scan cycle takes place in zero time. In comparison to the PLC the plant is so “slow” that it cannot distinguish a real PLC with scan cycles form an ideal time-continuous control. For a more detailed discussion of modelling PLCs see [11].

2.3 The control program

The goal of this section is to describe our view on the plant and the control program as we used it in an informal way. Its formal derivation and our other verification activities are presented in [12].

In the plant we identified a number of transport processes, such as transport of 4.2l salt solution from container B1 to B3. All possible transport processes, the evaporation process and two cooling process lead to a number of 12 parallel processes. The activities in each process are simply to open some valves, switch on a mixer, pump or heater, and when the process finishes, close and switch off everything again. Each process starts its activities if its *activation conditions* are fulfilled, and is in a wait state otherwise. An active process (state) remains active until its postconditions are fulfilled. Then it gets back in the waiting state. With this approach we have a so called *closed loop control*: the criterion to change a state is not that time proceeded, but an event occurs. The structure of the program is easy to find back in the SFC (Sequential Function Chart) representation in Figure 2.

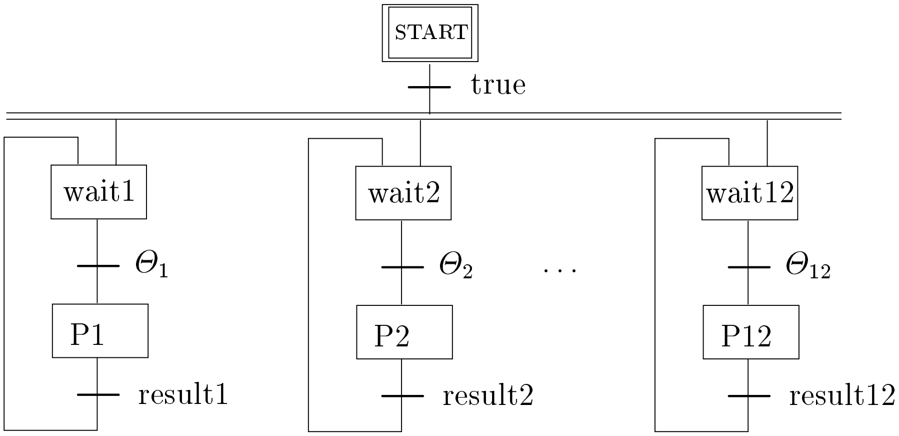

Fig. 2. The control program in Sequential Function Chart

Control starts in the state "START" and (because the transition condition is "true") immediately distributes to the 12 parallel wait states. In a wait state a process does nothing. If control comes to state Pi, the program attached to state Pi is executed in every scan cycle as long as control remains in Pi. The programs attached to P1, ..., P12 are contained in figure 3 in the so called Instruction List (IL) format. The instructions of IL are assembly-like. Here, we mainly load the constants true or false to the accumulator and write the accumulator value to the variables, e.g., Vi, representing the valves. The *action qualifier* P1 (at the top of each program) indicates that the instructions right to it are only executed in the first scan cycle when control is here; P0 says that the instructions are only executed in the last scan cycle when control is in this location.

The main complexity of the program is hidden in the activation conditions Θ_i. We assume to have a predicate Pi.X for each step Pi indicating whether control is at the corresponding step or not (these variables are available in PLC programs). The conditions to start a process (i.e. step) are infomally the following:

1. The filling levels of the tanks must allow for, e.g., a transport step: the upper tank must contain enough material, the lower tank must contain enough space, etc. These conditions are encoded in the predicates Φ_i of Figure 4.
2. We do not want a tank to be involved in two processes at a time. E.g., when transferring solution from B4 to B5 there should not be a concurrent transfer from B3 to B4. This requirement can be formulated by conditions on valves: when solution is transferred from B4 to B5 valve V11 must be closed for the duration of the transfer (invariant). These requirements induce a conflict structure on the processes. It is required that control is never at two conflicting processes at the same time. This condition is split into two parts: first, control cannot go to a process if a conflicting process is alraedy active.

These conditions are encoded in the predicates Ψ_i of Figure 5. Second, when conflicting processes could get control at the same moment only the one having *priority* gets it. These priorities are fixed, and their priority graph is cycle free. They induce the predicates Θ_i in figure 6.

P1 :

P1	LD true ST V8
P0	LD false ST V8

P2 :

P1	LD true ST V9
P0	LD false ST V9

P3 :

P1	LD true ST V8 ST Mixer
P0	LD false ST V8 ST Mixer

P4 :

P1	LD true ST V9 ST Mixer
P0	LD false ST V9 ST Mixer

P5 :

P1	LD true ST V11
P0	LD false ST V11

P6 :

P1	LD true ST V12
P0	LD false ST V12

P7 :

P1	LD true ST Heater
P0	LD false ST Heater

P8 :

P1	LD true ST V15
P0	LD false ST V15

P9 :

P1	LD true ST V17
P0	LD false ST V17

P10:

P1	LD true ST V29
P0	LD false ST V29

P11:

P1	LD true ST V18 ST V23 ST V22 ST V1 ST V3 ST Pump1
P0	LD false ST V18 ST V23 ST V22 ST V1 ST V3 ST Pump1

P12:

P1	LD true ST V20 ST V24 ST V25 ST V5 ST V6 ST Pump2
P0	LD false ST V20 ST V24 ST V25 ST V5 ST V6 ST Pump2

Fig. 3. Instruction List Programs for steps P1. . . . , P12

The execution mechanism of PLCs guarantees a synchronous execution of parallel steps: in each scan cycle each program attached to an active step is executed once. It is this synchronous mechanism that makes the conditions Θ_i to have the intended effect.

$$\begin{aligned}
\Phi_1 &:= (\text{B1} = \text{sol42C} \lor \text{B1} = \text{sol82C}) \land \text{B3} = \text{empty}\\
\Phi_2 &:= (\text{B2} = \text{water28C} \lor \text{B2} = \text{water56C}) \land \text{B3} = \text{empty}\\
\Phi_3 &:= (\text{B1} = \text{sol42C} \lor \text{B1} = \text{sol82C}) \land \text{B3} = \text{water28C}\\
\Phi_4 &:= (\text{B2} = \text{water28C} \lor \text{B2} = \text{water56C}) \land \text{B3} = \text{sol42C}\\
\Phi_5 &:= \text{B3} = \text{sol70C} \land (\text{B4} = \text{empty} \lor \text{B4} = \text{sol70C})\\
\Phi_6 &:= (\text{B4} = \text{sol70C} \lor \text{B4} = \text{sol140C}) \land \text{B5} = \text{empty}\\
\Phi_7 &:= \text{B5} = \text{sol70C} \land (\text{B6} = \text{empty} \lor \text{B6} = \text{water28C} \lor \text{B6} = \text{water28H})\\
\Phi_8 &:= \text{B5} = \text{sol42H} \land (\text{B7} = \text{empty} \lor \text{B7} = \text{sol42C} \lor \text{B7} = \text{sol42H})\\
\Phi_9 &:= \text{B7} = \text{sol42H} \lor \text{B7} = \text{sol84H}\\
\Phi_{10} &:= \text{B6} = \text{water28H} \lor \text{B6} = \text{water56H}\\
\Phi_{11} &:= (\text{B7} = \text{sol42C} \lor \text{B7} = \text{sol84C}) \land (\text{B1} = \text{empty} \lor \text{B1} = \text{sol42C})\\
\Phi_{12} &:= (\text{B6} = \text{water28C} \lor \text{B6} = \text{water56C}) \land (\text{B2} = \text{empty} \lor \text{B2} = \text{water28C})
\end{aligned}$$

Fig. 4. The tank filling conditions

$$\begin{aligned}
\Psi_1 &:= \Phi_1 \land \neg\, \text{P2.X} \land \neg\, \text{P4.X} \land \neg\, \text{P5.X} \land \neg\, \text{P11.X}\\
\Psi_2 &:= \Phi_2 \land \neg\, \text{P1.X} \land \neg\, \text{P3.X} \land \neg\, \text{P5. X} \land \neg\, \text{P12.X}\\
\Psi_3 &:= \Phi_3 \land \neg\, \text{P2.X} \land \neg\, \text{P4.X} \land \neg\, \text{P5.X} \land \neg\, \text{P11.X}\\
\Psi_4 &:= \Phi_4 \land \neg\, \text{P1.X} \land \neg\, \text{P3.X} \land \neg\, \text{P5. X} \land \neg\, \text{P12.X}\\
\Psi_5 &:= \Phi_5 \land \neg\, \text{P1.X} \land \neg\, \text{P2.X} \land \neg\, \text{P3.X} \land \neg\, \text{P4.X} \land \neg\, \text{P6.X}\\
\Psi_6 &:= \Phi_6 \land \neg\, \text{P5.X} \land \neg\, \text{P7.X} \land \neg\, \text{P8.X}\\
\Psi_7 &:= \Phi_7 \land \neg\, \text{P6.X} \land \neg\, \text{P8.X} \land \neg\, \text{P10.X} \land \neg\, \text{P12.X}\\
\Psi_8 &:= \Phi_8 \land \neg\, \text{P6.X} \land \neg\, \text{P7.X} \land \neg\, \text{P9.X} \land \neg\, \text{P11.X}\\
\Psi_9 &:= \Phi_9 \land \neg\, \text{P8.X} \land \neg\, \text{P11.X}\\
\Psi_{10} &:= \Phi_{10} \land \neg\, \text{P7.X} \land \neg\, \text{P12.X}\\
\Psi_{11} &:= \Phi_{11} \land \neg\, \text{P1.X} \land \neg\, \text{P3.X} \land \neg\, \text{P8.X} \land \neg\, \text{P9.X}\\
\Psi_{12} &:= \Phi_{12} \land \neg\, \text{P2.X} \land \neg\, \text{P4.X} \land \neg\, \text{P7.X} \land \neg\, \text{P10.X}
\end{aligned}$$

Fig. 5. A process may not start if a conflicting process is active

$$\begin{aligned}
\Theta_1 &:= \Psi_1 \land \neg\, \Psi_5\\
\Theta_2 &:= \Psi_2 \land \neg\, \Psi_1 \land \neg\, \Psi_3 \land \neg\, \Psi_5\\
\Theta_3 &:= \Psi_3 \land \neg\, \Psi_5\\
\Theta_4 &:= \Psi_4 \land \neg\, \Psi_1 \land \neg\, \Psi_3 \land \neg\, \Psi_5\\
\Theta_5 &:= \Psi_5 \land \neg\, \Psi_6\\
\Theta_6 &:= \Psi_6 \land \neg\, \Psi_7 \land \neg\, \Psi_8\\
\Theta_7 &:= \Psi_7\\
\Theta_8 &:= \Psi_8 \land \neg\, \Psi_7\\
\Theta_9 &:= \Psi_9 \land \neg\, \Psi_8\\
\Theta_{10} &:= \Psi_{10} \land \neg\, \Psi_7\\
\Theta_{11} &:= \Psi_{11} \land \neg\, \Psi_1 \land \neg\, \Psi_3 \land \neg\, \Psi_8 \land \neg\, \Psi_9\\
\Theta_{12} &:= \Psi_{12} \land \neg\, \Psi_2 \land \neg\, \Psi_4 \land \neg\, \Psi_7 \land \neg\, \Psi_{10}
\end{aligned}$$

Fig. 6. Of two conflicting processes only the one with priority may get active

3 Verification and optimization with Spin

This section describes our approach to the verification of the PLC program of Figure 2 and its subsequent optimisation. For the verification we constructed a model of the control program and the plant in Promela, while completely abstracting away from time. We used the model checker Spin to check that all execution sequences of the combined model satisfy the property that "always eventually batches are produced". This implies that under ideal circumstances, in which no material is lost through leakage or evaporation, control is such that new batches will always be produced. The details of the verification procedure are given in section 3.1; the technical conclusions are given in section 4.

To obtain also optimal schedules for the plant, in the sense that the average production time of a batch is minimal, we refined the Promela model by including light-weight real-time features. These sufficed find optimal scheduling sequences as counter-examples to properties stating suboptimal behaviour (cf. [4, 14]). This approach is described in more detail in section 3.2, with conclusions in section 4.

3.1 Correctness of the PLC program

Both the plant as described in section 2.1, and the informal control program description of section 2.3 can be translated into Promela in a straighforward way, the crucial part of the modelling exercise being the real-time properties of the plant in combination with the PLC execution mechanism given in section 2.2. In this case there are two basic principles that allow us to deal with the entire system by essentially abstracting away from time (see also [11] for a more general account in the context of PLCs):

1. The control program works independently of the time that the production steps take. Therefore, in the model each of the production steps P1, ..., P12 may take some unspecified time: if activated (e.g. by opening a valve) it goes to an undefined state that it eventually will leave to reach the final state where the result property holds. By this way of modeling every timing behaviour of the plant is subsumed, including the real one. If we can prove correctness for this general case, then correctness of the special case follows.
2. The excution speed of the control program is much faster than the tolerance of the plant processes, as was already mentioned above. This has two important implications:
   - we can abstract away from the scan cycle time and assume that scan cycles are executed instantaneously.
   - we can assume that the plant is continuously scanned so that state changes are detected without (significant) delay.

In our Promela model of the control program scan cycles are made instantaneous using the `atomic` construct. The model of the combined behaviour of the plant and the control program is obtained by putting the models of the control process and all the plant processes in parallel. Doing this, we must make sure that the continuous execution of the control program does not cause a starvation of the plant processes. This is taken care of by allowing only fair executions in Spin of our Promela model: in each execution no active process may be ignored indefinitely. We must be careful, however, not to lose the other important property, viz. that each state change of the plant is detected "immediately". Our model takes care of this by forcing a control program execution after each potential state change of the plant.

The Promela model of this case study is too big to be part of this paper. The full version can be retrieved from [2]. Here we present two excerpts, one of the plant model and one of the control program model, to illustrate its main features. Figure 7 contains the Promela process that models the transportation of solutions from container B1 to B3. It models the combined behaviour underlying steps S1 and S3.

The model consists of a do-loop that continuously tries to start the transfer of a unit of salt solution from B1 to B3 (corresponding to steps S1 and S3 of the specification). If the right conditions are fulfilled control will enter the body of the loop, and will mark the beginning of the transfer step by instantaneously (using the Promela `atomic` construct) changing the contents of both containers to undefined transitional states. At some later moment it will execute the second part of the body, instantaneously changing the transitional states to the corresponding terminal states, corresponding to the end of the transfer. Here, we have also added an assert statement between these two atomic statements, expressing an invariant that must always hold between the beginning and end of the tranfer step. As this may create a lot of extra states in the verification model this assertion can be removed to improve the performance when checking other properties.

Other observations that may help to understand this Promela model are:

- The `cycle` variable is a global flag that forces the excution of a scan cycle after the execution of each atomic step in the plant (flag is raised at the end of each such atomic step). After the execution of a scan cycle (also modelled as an atomic process, see below) the flag is lowered. Each atomic step in the plant is guarded by the test `cycle==0`.
- The Promela model combines steps in the plant that involve the same set of containers into one process. This reduces the number of processes that must be scheduled fairly.
- The Promela model of the plant models the transportation steps from a "physical" attitude and imposes fewer conditions for a transportation to take place that the formal plant specification in the corresponding steps. E.g. for transportation from B1 to B3 to take place it is only required that B1 is not empty and valve V8 is open.
- To compensate for this all illegal and unwanted states of the plant are explicitly modelled as error states (`error` is defined as `assert(false)`) whose

```
proctype B1toB3()
{   do
    :: atomic{ (cycle==0 && B1!=cempty && v8) ->
                  if
                  :: (B1==sol42C) -> B1=undef1
                  :: (B1==sol84C) -> B1=undef2
                  :: else -> error
                  fi ;
                  if
                  :: (B3==cempty) -> B3=undef1
                  :: (B3==water28C && mix) -> B3=undef2
                  :: else -> error
                  fi ;
                  cycle=1
        } ;
        assert(v8 && (B3!=undef2 || mix)) ;
        atomic{ (cycle==0 && v8) ->
                  if
                  :: (B1==undef1) -> B1=cempty
                  :: (B1==undef2) -> B1=sol42C
                  :: else -> error
                  fi ;
                  if
                  :: (B3==undef1) -> B3=sol42C
                  :: (B3==undef2 && mix) -> B3=sol70C
                  :: else -> error
                  fi ;
                  cycle=1
        }
    od
}
```

Fig. 7. The Promela model of transfer between B1 and B3

```
proctype Control()
{ int i,j ;
    do
    :: atomic{ i=1 ; j=1 ;
            do
            :: (i<15) ->
                if
                :: (theta(i,j) && !px[procnr(i)]) -> PB1(i)
                :: (result(i,j) && px[procnr(i)]) -> PB0(i)
                :: else -> skip
                fi ;
                if
                :: (j==1) -> j=2
                :: (j==2) -> j=1 ; i=i+1
                fi
            :: (i==15) -> goto endcycle
            od ;
            endcycle: cycle=0
        }
    od
}
```

Fig. 8. The Promela model of the control process

reachability can be checked. This approach gives us more information about the robustness of our controller.

The Promela process that models the control program is listed in Figure 8. This is a straightforward translation of the PLC program of Figure 3.

The do loop of `Control` repeatedly executes an atomic scan cycle, in which the processes P1,. . . ,P12 are scheduled sequentially. To deal with the symmetric subcases of each step (i.e. the disjuncts between brackets in Figure 4) we need a second loop counter `j` next to the main counter `i` (because P11 and P12 in fact have 4 subcases P11 is covered by `i`$\in \{11, 12\}$ and `j`$\in \{1, 2\}$, and P12 by `i`$\in \{13, 14\}$ and `j`$\in \{1, 2\}$). Modulo these small adaptations the `theta(i,j)` correspond to the Θ-predicates of Figure 6, and the `result(i,j)` correspond to analoguous formalisation of result conditions of the PLC program (the uninstantiated result$i$ labels of Figure 2). `PB1(i)` and `PB2(i)` correspond to the code of the P1 part, and the P0 part of the PLC program, respectively. The variables `px[i]` correspond to the Pi.X activity predicates of the program mentioned earlier. Note that at the end of each scan cycle the global flag `cycle` is lowered, as required.

Whereas the assertions in the model served to check on our own understanding the model, the main correctness requirement that "always eventually a new batch is produced" was verified using the Spin facilities for model checking LTL formulas. The requirement was formalized as the following LTL property:

$$\Box \Diamond \texttt{(B3 == sol70C)} \wedge \Box \Diamond \texttt{(B3 == cempty)} \tag{1}$$

expressing that the contents of container `B3` (containing the brine solution that is considered the "production" of the plant) is infinitely often full and infinitely oftem empty (the constant `cempty` was chosen to be different from the Promela reserved word `empty`). As these two properties must interleave in each linear execution sequence they are equivalent to the desired requirement.

It turned out to be feasible to run the model checker sequentially on our model initialised with material for 0 up to 8 batches (including the intermediate different possibilities for half batches; 30 runs in total). In order to avoid the explosion of the more than 8100 possible initial configurations that are in principle possible, we considered only configurations filling the plant "from the top", i.e. filling tanks in the order B1,. . . ,B7. The other initializations are reachable from these by normal operation of the plant. As satisfaction of the property that we checked (see below) for our initial configurations implies its satisfaction for all reachable configurations this is sufficient. Using simulations of our model we satisfied ourselves that our model did include the required normal operation steps (here, model checking would run into the same combinatorial explosion).

After initial simulations and model checking runs had been used to remove small (mainly syntactic) mistakes from our model, the model was systematically checked for property (1) for the 30 initializations with different batch volumes described above. No errors were reported, except for initializations with batch volumes 0, 0.5, 7.5 and 8, as should be the case. The model checking was done using Spin version 3.3.7 on a SUN Enterprise E3500-server (6 SPARC cpus with 3.0 GB main memory). The model checking was run in exhaustive state space search mode with fair scheduling. The error states reported unreachable in all runs. The shortest runs were completed in the order of seconds and consumed in the order of 20MB memory; the longest run required in the order of 40 minutes and 100MB.

3.2 Deriving optimal schedules

The control schedule of Figure 2 that we have shown to be correct by the procedure sketched in the previous subsection, follows an essentially crude strategy. After each scan cycle it enables *all* non-conflicting processes in the plant whose preconditions it has evaluated to hold true. It is not a priori clear that this strategy would also lead to a plant operation that is optimal in the sense that the average time to produce a batch is minimal.

To determine optimal schedules for the various batch loads of the plant we have refined the models of the previous section as follows:

1. We added a notion of time to the model. To avoid an unnecessary blow-up of the state space due to irrelevant points in time, i.e. times at which nothing interesting can happen, we have borrowed an idea from discrete event simulation, viz. that of *variable time advance procedures* [15].
2. We refined the plant model using the information from Table 1, such that each process in the plant will take precisely the amount of time specified.
3. We refined the model of the control program such that after each scan cycle any non-empty subset of the maximal set of allowed non-conflicting processes determined by the original control program could be enabled.

The search for optimal schedules was then conducted by finding counterexamples for the claim:

$$\Box(\texttt{batches} < \texttt{N}) \tag{2}$$

where `batches` is a global variable that counts the number of times that a brine solution is transferred from `B3` to `B4`. This property is checked for increasing values of `N` in the context of a given maximal clock value `maxtime`. The assumption is that for `maxtime` large enough such counterexamples will display regular scheduling patterns. Below, we elaborate on each of the above points and the search procedure.

A variable time advance procedure In real-time discrete event systems events have associated clocks that can be set by the occurrence of other events.

An event occurs when its clock expires. Such systems can be simulated by calculating at each event occurrence the point in time at which the *next* event will occur, and then jumping to that point in time. This is known as *variable time advance*.

We wish to apply this idea to our model because it will not litter the global state space with states whose time component is uninteresting, in the sense that there is no process in the plant that begins or ends. As we can only calculate when plant processes will end once they have started, we can only use this time advance procedure if we assume that processes will always be started when others end (or at time 0). It is not difficult to see, however, that we will not lose schedules this way that are strictly faster than what we can obtain using this policy. The informal argument is as follows: assume that a derived scheduling policy can be strictly improved by postponing a given event e by some time t. Because we are optimising w.r.t. time (and not energy or resource utilisation or the like), the more optimal schedule must exploit this time to start a *conflicting* process (ending a conflicting process would have prevented e in the original schedule; any event associated with a non-conflicting process can be executed anyway). Because this process is conflicting it must also finish before e occurs. We may therefore assume that in any optimal schedule e is excuted when the last preceding conflicting process ends.

The variable time advance procedure is implemented by the Promela process `Advance` given in Figure 9. The basic idea of `Advance` is quite simple: when it becomes active it will calculate the next point in time when a plant process will terminate. To do so it uses the global array `ptime(i)` containing the termination times of the processes `i`, whose values are calculated as part of the Promela processes modelling the plant, and the global time variable `time`, which is controlled by `Advance`. `maxstep` is a global constant corresponding to the longest possible time step in the model, i.e. the duration of the heating process. All variables related to time are of type `short`, a unit corresponding to 10 seconds in Table 1 as al its entries are multiples of 10 seconds. `Advance` will be activated only when the predicate `promptcondition` holds. This predicate is true if and only if all processses that have been enabled by the control program have indeed become active and none has terminated.

The refined plant model The refined model of the plant differs from the original model in the folowing respects:

- The (atomic) start event of each plant process is used to calculate the termination time of that process.
- The termination event of each plant process is guarded with the additional condition that the global time `time` must equal the calculated termination time.
- The start and termination events include `printf` statements to record activation and termination times to enable the analysis of simulated executions (of the counterexample trails).

```
proctype Advance()
{ int i ; short minstep ;
    do
    :: atomic{(promptcondition) ->
            minstep=maxstep ; i=1 ;
            do
            :: (i<13) ->
                if
                :: (px[i] && ((ptime(i)-time)<minstep)) ->
                    minstep=(ptime(i)-time)
                :: else -> skip
                fi ;
                i=i+1
            :: (i==13) -> goto step
            od ;
            step: time=time+minstep
        }
    od
}
```

Fig. 9. The Promela model of the time advance process

The refined control model To allow the new model of the control program to enable any nonempty subset of the permissable plant process start events, we have split the loop of the original model of Figure 8, resulting in Figure 10. The first of the two loops scans only for termination conditions of plant processes and executes the corresponding control fragments `PB0(i)`. The second loop subsequently scans the valid preconditions of the plant processes. The corresponding control fragments `PB1(i)` may or may not be executed. If not, the process number is stored in the local variable `last`, possibly overwriting a previous value. If the second loop is exited without any processes being active (`act` is false), then the process with number `last` is activated.

The idea to retrospectively activate the last plant process that could have been activated to prevent the plant from becoming inactive, cannot be implemented in the original, single control loop. There, plant process terminations occuring after the evaluation of the precondition corresponding to `last` could invalidate the precondition, rendering subsequent activation impossible.

Both loops of the new version are contained in a new outer loop that monitors the progress of time and will stop control if `time` exceeds `maxtime`. This will cause the combined plant and control model to terminate.

Finding optimal schedules Finding optimal schedules we restricted ourselves to the interesting cases involving initial plant loads of 1 through 7 batches. For our initial experiments we fixed `maxtime` to be 5000 time units (50,000 s). For each initial load we needed two or three runs to determine the maximal number of batches for which counterexamples could be produced in a very short time (in the order of seconds real time). It turned out that all counterexamples produced contained schedules that rapidly (i.e. within 300 time units) converged to a repeating pattern with a fixed duration.

The initial measurements are collected in Table 2. The interpretation of the columns is as follows:

- load: indicates the number of batches with with the plant is initialised,
- simtime: indicates the the duration (in simulated time units) of the counterexample traces,
- batches: the number of batches produced in that trace,
- states: the number of states visited to produce the trace,
- transitions: the number of transitions visited to produce the trace,
- convergence: the convergence time until periodic behaviour,
- period: period time of periodic behaviour.

A first analysis of Table 2 shows the state space that needs to be searched to produce the counterexamples is very small, and could make one suspicious of the quality of the results that are obtained. Surprisingly enough, five of the measured periods turn out to be optimal schedules! For loads with 1 and 7 batches this can be readily checked by hand by moving a single batch through the plant, or the empty space for a batch (the total volume of the plant is 8 batches), respectively, and measuring the total duration of the critical branches of the path.

Initially, we thought that we had made a mistake when we measured the same period of 173 units for loads 2, 3 and 4. Closer analysis of the schedules, however, revealed that this is the result of the fact that the plant has one process that clearly dominates the time consumption during the production of batches, viz. the heating of container B5 (110 time units). Since filling B5, heating it, and emptying B5 must be part of every production cycle, the average production time of a batch must be greater or equal then 35+110+28=173 time units. This makes the schedules underlying the period of 173 for loads 2, 3 and 4 optimal schedules as well.

load	simtime	batches	states	transitions	convergence	period
1	4767	17	1185	1510	56	294
2	4682	28	1916	2450	56	173
3	4972	31	2063	2639	294	173
4	4886	30	2031	2598	208	173
5	3761	20	1449	1866	208	197
6	3885	20	1567	2072	173	195
7	4340	17	1202	1532	173	260

Table 2. Initial schedule measurements

The previous observation made us think that schedules for loads 5 and 6 could be improved upon, as they are in some sense dual to the cases for loads 2 and 3 (moving empty batch space upwards through the plant instead of batches downwards). In fact, inspection of the counterexample for load 6 clearly showed

```
proctype Control()
{   int i,j,last ; bool precon, postcon ;
    do
    :: (time<maxtime) ->
        atomic{i=1 ; j=1 ;
            do
            :: (i<15) ->
                postcon=(result(i,j) && px[procnr(i)]) ;
                if
                :: postcon -> PB0(i)
                :: else -> skip
                fi ;
                if
                :: (j==1) -> j=2
                :: (j==2) -> j=1 ; i=i+1
                fi
            :: (i==15) -> goto loop2
            od ;
            loop2:
            i=1 ; j=1 ; last=1 ;
            do
            :: (i<15) ->
                precon=(theta(i,j) && !px[procnr(i)]) ;
                if
                :: precon -> PB1(i)
                :: precon -> last=i
                :: else -> skip
                fi ;
                if
                :: (j==1) -> j=2
                :: (j==2) -> j=1 ; i=i+1
                fi
            :: (i==15) -> goto finish
            od ;
            finish:
            if
            :: (!act) -> PB1(last)
            :: else -> skip
            fi ;
            cycle=0
        }
    :: (time>=maxtime) -> goto endtime
    od ;
    endtime: skip
}
```

Fig. 10. The refined model of the control process

that it could be improved. As increasing the number of batches immediately led to a dramatic increase of the response time for producing counterexamples, we looked for cheaper ways to get feedback more quickly. There are two dimensions that determine the state space to be explored, viz. the depth of the search tree and its branching degree. The first can be made smaller by reducing the value of `maxtime`, the second by exploring fewer scheduling alternatives in the control program.

For the second option we had the original control schedule of Figure 8 at our disposal. This process does not lead to a completely deterministic scheduling, because it may make a difference whether the scan cycle is executed once or more often between plant events. This is because the termination of some processes later in the scan cycle may enable the beginning of other plant processes earlier in the (next) scan cycle. The result therefore stabilises after two scan cycles. Using this much leaner search tree we did in fact find optimal schedules for loads 5 and 6, again with period 173, in a matter of seconds, see Table 3.

load	simtime	batches	states	transitions	convergence	period
5	3329	20	1380	1761	35	173
6	3467	20	1415	1806	173	173

Table 3. Measurements for loads 5 and 6 with the original control program

Also the first option, reducing the search tree by reducing `maxtime`, can be used with success. Reducing the simulated time to 519 time units, we could find an optimal schedule for a system load of 6 producing 3 batches. This option required a more extensive search, however, involving more than 4.5 million states, showing that the optimal schedule here is contained in a part of the tree explored much later than in the previous examples. We did not systematically apply this approach to the other loads.

It must be concluded that the plant can be scheduled in the overall optimal time of 1730 seconds for all loads, except for the extreme loads of 1 and 7. Because of our analysis above, these are not only time optimal but also resource optimal schedules, in the sense that the (expensive) destillation container `B5` is in continuous use. From the energy perspective, probably the schedule for load 2 is optimal, as this involves the circulation, heating and cooling of the smallest volume.

4 Conclusions

In this paper we have shown how the Promela/Spin environment can be used to verify and optimize control schedules for a small-size PLC controlled batch plant. The approach in this paper relies quite heavily on the structured design of an initial control program that can be found in [12] and the analysis of formal approaches to PLCs in [11].

It is interesting to see that we succeeded in dealing with this real-time embedded system using standard Promela/Spin. For the verification of the initial control program this was due to a property of the plant, viz. that we could assume instantaneous and immediate scanning of all state changes of the plant. This as a consequence of the tolerance of the plant processes for much slower reaction times than those realised by the PLC control. This makes us conclude that this abstraction can be used for checking non-timed correctness criteria in all process control problems that have this property.

The original task we set ourselves was just to check the correctness of the plant control in the sense that the designed program would in principle always be capable of producing more batches for any reasonable initial load. Having achieved that task we wondered how the model might be used to also look at the optimality of the schedules. As we wanted to treat this in terms of small modifications of the model only, we added time in the form of an explicit time advancing process. This is very close in spirit to the real-time Promela/Spin extension DTSpin [1]. Given the particular properties of the plant, however, viz. that without loss of optimality plant processes can be assumed to start when others terminate, we could do this by only generating those points in time in which plant events could take place. From the schedules that we obtained we can conclude that in this case study this variable time advance procedure reduced the generated state space by approximately a factor of 20.

On the basis of our modified model we could find optimal schedules surprisingly quickly. This is certainly due to the particular characteristics of the given plant, with its very critical heating process. Also, we have been lucky in the sense that the optimal schedules often were found in those parts of the search tree that were explored earlier. Counterexamples were produced so quickly, in fact, that the gain of the factor of 20 by using the time advance procedure seemed immaterial. There is one exception, however, viz. searching the optimal schedule for load 6 using the refined (nondeterministic) `Control` process. By drastically reducing `maxtime` we obtained an optimal schedule while storing some 4.5 million states. Given the 132 byte state vector, in this case the reduction factor of 20 appears very useful. Although more experiments are certainly needed, we believe that variable time advance procedures can be useful for this kind of application. One way to think of them is as an explicitly programmed analogon of the notion of time regions as in timed automata [3]. Taking advantage of specific properties of systems such as ours an explicit approach can sometimes yield better results.

To apply our technique for finding optimal schedules, viz. by generating counterexamples for claims of suboptimal behaviour, in more general cases, it would be useful to be able to influence the search strategy of the model checker more directly and guide the search first into those parts of the search trees where counterexamples are likely to be found. [5] discusses how branch and bound algorithms could be used for such purposes, especially in the context of model checking for timed automata (UPPAAL [10]). Our results indicate that it can be wortwhile to investigate such guided search methods also for non-real time model checkers like Spin.

Another study of the optimal scheduling for the VHS case study 1 is reported in [13]. Here the problem is analysed using the tools OpenKronos and SMI. It is difficult to compare the results of this approach directly with ours, as they include also the production of the initial loads into their schedules, which we just assume to be present. The more general findings seem to be consistent with ours, however. OpenKronos could be used succesfully to produce optimal schedules for loads of up to 3 batches before falling victim to the state explosion problem. The symbolic model checker SMI produced results 6 batches and more, with a computation time of approximately 17 minutes per batch.

References

1. Dtspin homepage. `http://www.win.tue.nl/d̃ragan/DTSpin.html`.
2. VHS: Case study 1 sources. `http://www.cs.kun.nl/m̃ader/vhs/cs1.html`.
3. R. Alur and D.L. Dill. A theory of timed automata. *Th. Computer Science*, (138):183–335, 1994.
4. A. Fehnker. Scheduling a steel plant with timed automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.
5. A. Fehnker. Bounding and heuristics in forward reachability algorithms. Technical Report CSI-R0002, University of Nijmegen, Netherlands, February 2000.
6. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, (1):110–123, 1997.
7. G.J. Holzmann. The model cheker SPIN. *IEEE Trans. on Software Eng.*, 23(5):279–295, May 1997.
8. International Electrotechnical Commission. *IEC International Standard 1131-3, Programmable Controllers, Part 3, Programming Languages*, 1993.
9. S. Kowalewski. Description of case study cs1 "experimental batch plant". `http://www-verimag. imag.fr/VHS/main.html`, July 1998.
10. K.G. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, (1):134–153, 1997.
11. A. Mader. A classification of PLC models and applications. submitted to WODES, 2000.
12. A. Mader, E. Brinksma, H. Wupper, and N. Bauer. Design of a plc control program for a batch plant - vhs case study 1. submitted for publication, `http://www.cs.kun.nl/˜ mader/papers.html`, 2000.
13. Peter Niebert and Sergio Yovine. Computing optimal operation schemes for multi batch operation of chemical plants. VHS deliverable, May 1999. `http://www-verimag.imag.fr/VHS/main.html`.
14. Th. Ruys and E. Brinksma. Experience with literate programming in the modelling and validation of systems. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 393–408. Springer-Verlag, 1998.
15. G.S. Shedler. *Regenerative Stochastic Simulation*. Academic Press, 1993.
16. F.W. Vaandrager and J.H. van Schuppen. *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
17. S. Yovine. Kronos: a verification tool for real-time systems. *Software Tools for Technology Transfer*, (1):123–134, 1997.