Int J STTT (2002) 4: 21–33 / Digital Object Identifier (DOI) 10.1007/s10009-002-0079-0

Verification and optimization of a PLC control schedule

Ed Brinksma[1], **Angelika Mader**[1], **Ansgar Fehnker**[2]

[1] Faculty of Computer Science, University of Twente, Netherlands; E-mail: {brinksma,mader}@cs.utwente.nl
[2] Electrical & Computer Engineering, Carnegie Mellon University, USA; E-mail: ansgar@ece.cmu.edu

Published online: 2 October 2002 –

Abstract. We report on the use of model checking techniques for both the verification of a process control program and the derivation of optimal control schedules. Most of this work has been carried out as part of a case study for the EU 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 original 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*. To compare and interpret the results we carried out the same case study with the aid of the real-time model checker UPPAAL, enhanced with facilities for cost-guided state space exploration. Both approaches proved sufficiently powerful to verify the design of the controller and/or derive (time-)optimal schedules within reasonable time and space requirements.

Keywords: Formal methods – Verification – Model checking – Hybrid systems – Scheduling

1 Introduction

The verification of hybrid systems is a research area of rapidly growing importance 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 [7, 11, 17].

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 Controllers or PLCs. Unfortunately, both PLCs and their associated programming languages have no well-defined formal models or 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 EU 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 article we report on the use of two model checkers for the verification of a process control program for the given plant and the derivation of optimal control schedules. It is a companion paper to [13], which concentrates on the correct design of the process controller. The original intention of our approach was to see how much could be achieved using the standard model checking environment of SPIN/Promela [8]. 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

The work reported here was carried out while the second and third authors were employed by the Computer Science Department of the University of Nijmegen, Netherlands. The second author was supported by an NWO postdoc grant, the third author by an NWO PhD grant, and both were supported by the EU LTR project VHS (Project No. 26270).

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. A first report on our experiences has been published as [5], whose findings are further extended and elaborated in this article.

One of the conclusions of our initial experiments as reported in the initial publication [5] was that "... 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 ... where counterexamples are likely to be found." Since this publication a new version of the real-time model checking tool UPPAAL has become available that employs a cost-guided evaluation strategy for state-space exploration, viz., *cost-optimal* UPPAAL [4]. This tool is a natural candidate to support the derivation of optimal control schedules in a real-time environment. This motivated us to carry out the optimization part of the case study again with cost-optimal UPPAAL, both as an interesting exercise in its own right, and to collect data to interpret and compare with the results obtained with SPIN.

The rest of this paper is organized as follows: Sect. 2 gives a description of the batch plant, the nature of PLCs, and a description of the control program that was systematically designed in previous work [13]. Section 3 describes the Promela models for the plant and the control process, and their use for its formal verification and optimization. Section 4 then introduces a cost-optimal UPPAAL model of the same processes, and presents the optimization results that were obtained using it. Section 5, finally, evaluates the work and presents our 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 in the first place that new batches can always be produced, and, in the second place, optimality of the control schedule.

2.1 Description of the batch plant

The batch plant (see Fig. 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 Kowalewski's description of the plant [10].

It "produces" batches of diluted salt solutions from concentrated salt solutions (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: it consists of either a volume of 4.2 l salt solution with a concentration of 5 g/l and a volume of 2.8 l pure water distributed over two containers, or, when mixed, a volume of 7 l salt solution of 3 g/l concentration in a single container. With these batch sizes containers B1, B2, B4, B6, and B7 have a capacity of two "units" of the relevant volumes, and B3 and B5 of only one such "unit". The plant description [10] gives 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.[1]

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* (see Fig. 2). In each scan cycle the program in the PLC is executed once, where the program execution may depend on variable 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, a part of each scan cycle is dedicated to 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

[1] Note that these durations, given in Table 1, differ from those used in the initial report [5], which were chosen to be more comparable to the figures used in similar work by Niebert and Yovine [14].

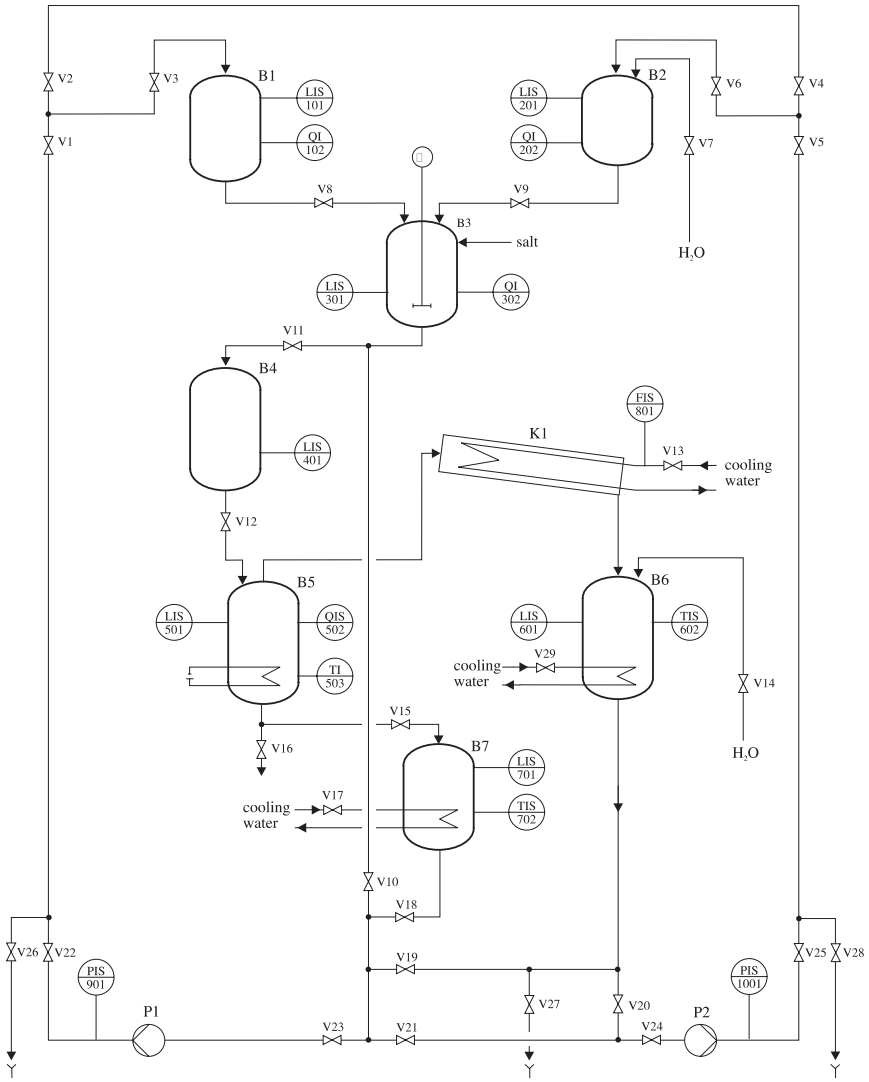

Fig. 1. The P/I-diagram of the batch plant

Table 1. Duration of plant processes in seconds

B1–B3	B2–B3	B3–B4	B4–B5	heat B5	B5–B7	cool B6	cool B7	B6–B2	B7–B1
320	240	600	330	1470	260	300	600	240	220

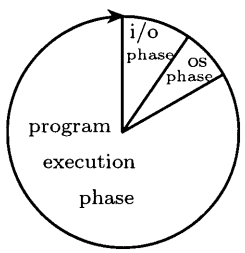

Fig. 2. A PLC scan cycle

output points the PLC sends data to actuators. Finally, there are some activities of the operating system (self checks, watch-dogs, etc.) that take place in a scan cycle. The operation 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 down-loaded to the PLC.

There are different domain-specific programming languages collected in an IEC standard for PLCs [9]. 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 that moment. SFC thus provides for the higher-level structure and the actual instructions of our application are written in Instruction List (IL), an assembly-like language. The PLC languages offer *timers*

as basic constructs, which also differs from most of the usual programming languages.

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 from an ideal time-continuous control. For a more detailed discussion of modelling PLCs see [12].

2.3 The control program

This section gives an informal description of the control program as we used it in our verification and optimization exercises. Its formal derivation and a description of our other related verification activities can be found in a previous publication [13].

In the plant we can identify a number of basic plant processes, such as the transport of 4.2 l salt solution from container B1 to B3. All possible transport processes, the evaporation process, and two cooling processes lead to 12 basic processes, which run in parallel. The activities of each process are simply to open some valves, switch on a mixer, pump or heater, and when the process is finished, close and switch off everything again. Each process starts its activities if its *activation conditions* are fulfilled, and otherwise is in a wait state. An active process remains active until its postconditions are fulfilled. Then it goes back into its waiting state. This means that we have a so-called *closed loop control*: the criterion to change state is not that sufficient time has elapsed, but that a particular event occurs.

It is not difficult to recognize this structure in the SFC representation of the program given in Fig. 3. Control starts in the state "START" and (because the transition condition is "true") immediately distributes to the 12 parallel wait states, waiting for the corresponding *activation conditions* θ_i to become true. In a wait state a process does nothing, i.e., it executes an empty program. If the activation condition becomes true, control reaches control state Pi. The program associated with Pi is executed in every scan cycle for as long as control remains in Pi. The IL programs P1, ..., P12 are given in Fig. 4. The instructions of IL are assembler-like. Here, we mainly load the constants true or false into the accumulator and write the accumulator value to one of the variables, e.g., Vi, representing valve number i. The *action qualifiers*[2] P1 and P0 (to the left of each program) indicate when the corresponding program blocks are executed. P1-labelled code is only executed in the first scan cycle after the control location has been reached; P0 indicates that the instructions are only executed in the last scan cycle that the control is at this location, i.e., when the postcondition evaluates to true – one last scan cycle is executed *after* the postcondition has become true.

The main complexity of the program is hidden in the design of the activation conditions Θ_i. We can assume to have predicates 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) can now be stated informally as follows:

1. The filling levels of the tanks must allow, for example, for a transport step: the upper tank must contain

[2] There exist more action qualifiers, e.g., to express that code must be executed at every scan cycle; in our case they are not needed for the control program code.

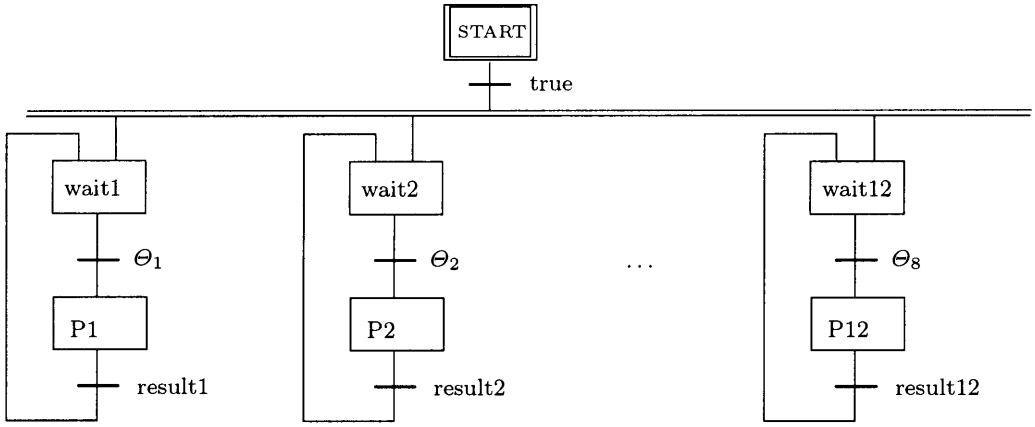

Fig. 3. The basic control program in Sequential Function Chart

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. 4. Instruction List Programs for steps P1. . . . , P12

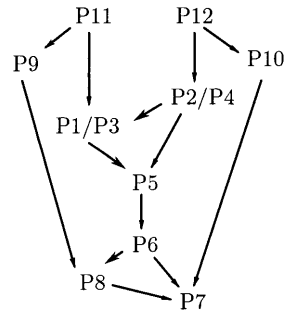

Fig. 5. Priority graph of the processes P1, . . . , P12; an arrow indicates that there is a conflict between processes, where arrows point to the processes with priority

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

Fig. 6. The tank filling conditions

enough material, the lower tank must contain enough space, etc. These conditions are encoded in the predicates Φ_i of Fig. 6.

2. We do not want a tank to be involved in two (or more) processes at a time. For example, 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 the conflict structure on the processes in Fig. 5.

   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 already active. These conditions are encoded in the predicates Ψ_i of Fig. 7. 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 in Fig. 5 is cycle free. They induce the predicates Θ_i in Fig. 8.

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

3 Verification and optimization with SPIN

To verify the correctness of the PLC program of Fig. 3 we made a Promela model of the plant and the control program, and used SPIN to check that all execution sequences of their composition satisfy the property that "always eventually batches are produced". This implies that under ideal circumstances in which no material is

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

Fig. 7. Activation conditions testing for active conflicting processes

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

Fig. 8. Predicates encoding the priority between conflicting processes; there are (non-circular) evaluation dependencies

lost through leakage or evaporation, control is such that new batches will always be produced. Subsequently, we refined the models to find more sophisticated variations of the control program that are time-optimal, in the sense that the average production time of a batch is minimal. These optimal scheduling sequences were produced as counter-examples to properties stating suboptimal behaviour. The details of these exercises are given below.

3.1 Correctness of the PLC program

Both the plant as described in Sect. 2.1, and the informal control program description of Sect. 2.3 can be translated into Promela in a straightforward 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 Sect. 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 [12] for a more general account in the context of PLCs):

1. The control program is independent of the time that the production steps take. In the model each of the production steps P1, . . . , P12 may take some unspecified time: when activated (e.g., by opening a valve) the control enters an 'undefined' state that will eventually be left to reach a final state once the corresponding postcondition holds. In this way every consistent real-time behaviour of the plant is subsumed, including the real one. Proving correctness for the general case implies correctness of the real case.
2. The execution 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 scanned continuously so that state changes are detected without (significant) delay.

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 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 the VHS document repository [2]. Here we present two excerpts, one of the plant model and one of the control program model, to illustrate the main features.

Figure 9 contains the Promela process that models the transfer of solution from container B1 to B3. It combines the behaviours underlying steps P1 and P3. The model consists of a do-loop that continuously tries to start the transfer of a unit of salt solution from B1 to B3. 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, marking the end of the transfer. [3] The `cycle` variable is a global flag that forces the execution 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

[3] Here, and in other parts of the Promela model, we have initially added `assert` statements that express invariants that should hold at the corresponding control location (`serror` being shorthand for `assert(false)`). These are used to check for the correctness of the model in initial simulation and verification trials, and are removed during the actual verification to avoid unnecessary growth of the state space.

```
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. 9. The Promela model of transfer between B1 and B3

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 process that models the control program is listed in Fig. 10. This is a straightforward translation of the PLC program of Fig. 4. 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 sub-cases of each step (i.e., the disjuncts between parentheses in Fig. 6) the range of the main counter is extended and a second loop counter `j` is introduced. Modulo these small adaptations the Θ-predicates of Fig. 8 are captured by the `theta(i,j)`, and the `result(i,j)` correspond to formalizations of the result conditions of the PLC program (the uninstantiated result$i$ labels of Fig. 3). `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 of 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. As a correctly operating plant is constantly recycling the same volume of batch material in this case the notion of when a batch is being produced is not completely straightforward. We have interpreted the

```
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. 10. The Promela model of the control process

production of diluted salt solution in container `B3` as the production of a batch, but other choices would have been equally defensible (e.g., storing a unit of this solution in `B4`, etc.). The correctness requirement was formalized as the following LTL property:

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

expressing that the contents of container `B3` is infinitely often filled with the diluted salt solution, and infinitely often empty[4]. Inspection of the Promela model for processes involving B3 shows that there are two possible execution sequences:

1. B3 is empty, then filled with concentrated solution from B1, then mixed with water from B2 giving the desired diluted solution (`sol70C`), and finally being emptied again, or
2. B3 is empty, then filled with water from B2, then mixed with concentrated solution from B1 giving the desired diluted solution (`sol70C`), and finally being emptied again.

As the two constituent properties of (1) must both be fulfilled, neither of the above scenarios is allowed to get stuck at some state, making the statement equivalent to the desired requirement that new batches of diluted solution are always eventually produced.

Verification results. It turned out to be feasible to apply the model checker sequentially to the different initializations of our model with material from zero up to eight 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

[4] The constant `cempty` was chosen to be different from the Promela reserved word `empty`.

are reachable from these by normal operation of the plant. As satisfaction of correctness property (1) for our initial configurations implies its satisfaction for all reachable configurations, this is sufficient. Using simulations of our model we convinced ourselves that our model did indeed include the required normal operation steps.

After initial simulations and model checking runs had been used to remove small mistakes from our model (identifier overloading, initialization errors), the model was systematically checked for property (1) with the 30 different initializations with 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, since the plant can only be productive if at least one batch volume can be circulated. 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 20 MB memory; the longest run required in the order of 40 MB and 100 MB.

3.2 Deriving optimal schedules

The control schedule of Fig. 3 that we have shown to be correct by the procedure sketched in the previous section, 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 to introduce scheduling alternatives for the control program. This was done by a nondeterministic choice selecting between subsets of the maximal set of allowed non-conflicting processes, as determined by the original control program.

The search for optimal schedules was conducted by finding counterexamples to 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`, i.e., no events will be possible when time exceeds the value of `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* [15].

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 then what we can obtain using this policy.

Claim. For each schedule s there exists a schedule s' in which no scheduled event occurs later and processes are exclusively started at the beginning of the schedule or when other processes end.

Proof sketch. Assume that s is a schedule. If s has the desired format take $s' = s$, otherwise there exists in s a first scheduling of a process p with a start event b_p and a termination event e_p such that the occurrence of b_p does not coincide with a termination event of another process p'. Let t be the latest point in time of a process termination occurrence before b_p, or, if no such occurrence exists, the start time of the schedule. Any process that conflicts with p must have terminated at or before t, or started at or after $s(e_p)$ to avoid overlapping with p, where $s(e)$ denotes the time at which event e is scheduled according to s. However, then we can produce a new schedule ns by putting $ns(b_p) = t$, $ns(e_p) = s(e_p) - (s(b_p) - ns(b_p))$ and $ns(e) = s(e)$ for all other events. If ns has the desired format take $s' = ns$, otherwise repeat the same procedure starting with ns. Because processes do not have arbitrarily small processing times (i.e., they are non-Zeno), the desired schedule s' is obtained in the limit.

The variable time advance procedure is implemented by the Promela process `Advance` given in Fig. 11. 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

```
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. 11. The Promela model of the time advance process

Promela processes modelling the plant, and the global time variable `time`, which is controlled by `Advance`. The activation of `Advance` is controlled by the predicate `promptcondition`. This predicate is true if and only if all processes that have been enabled by the control program have indeed become active and none have yet terminated.

The refined plant model. The refined model of the plant differs from the original model by adding some simple timing information to the plant processes. 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 then guarded with the additional condition that the global time `time` must equal the calculated termination time.

The refined control model. To allow the new model of the control program to enable any subset of the permissible plant process events at any time, we split the loop of the original model of Fig. 10. The first of the two loops scans only for termination conditions of plant processes and executes the corresponding control fragments `PB0(i)`. Subsequently, any of the conditions Ψ_i of Fig. 7 can be set nondeterministically to the value `false` before the second loop is entered. This loop scans the remaining valid preconditions of the plant processes of which the corresponding control fragments `PB1(i)` are then executed. All guards in both loops of the new version contain tests to monitor the progress of time and will stop control if `time` exceeds `maxtime`. This will cause the combined plant and control model to terminate.

The refinement as described above causes an exponential blow-up of the branching structure of the control process, as maximal sets of non-conflicting events are replaced by a choice between all their subsets. This leads to a corresponding blow-up of the exploration time for the state space, even if no new states are reached in this way. To control this phenomenon a global system parameter `cuts` is introduced that specifies the maximal number of events per branching point that may be postponed when enabled. Without the use of this parameter the state space exceeded our resources in practically all experiments (> 1 GB memory, > 40 min response time).

Finding optimal schedules. Looking for optimal schedules we restricted ourselves to the interesting cases involving initial plant loads of one through seven 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 system time). It turned out that all counterexamples produced contained schedules that rapidly (i.e., within 700 time units) converged to a repeating pattern with a fixed duration. None of the counterexamples required a `cuts` value greater than 2.

The best measurements in terms of shortest periods detected, are collected in Table 2. The interpretation of the columns is as follows:

load: indicates the number of batches with which the plant is initialized,
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,
steps: the number of atomic steps executed,
period: period of the periodic behaviour in time units.

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. Here, one should realize that the state space of the model has been kept small by the consequent use of `atomic` constructs for large Promela code chunks, the use of variable time advance, and the fact that the reachable part of the plant model state space is not so large (e.g., it does not have complicated data structures or queues). As the column with the atomic step counts indicates the small number of states does not make it a trivial model checking problem. This is caused by the substantial nondeterministic branching in the model, where many alter-

Table 2. SPIN schedule measurements

load	simtime	batches	states	steps	period
1	3476	10	711	94 253	380
2	4968	25	1715	239 277	206
3	4732	25	1652	230 236	206
4	4774	25	1736	241 261	206
5	3947	20	1423	196 627	206
6	4120	20	1459	201 685	206
7	3320	10	1935	2 568 780	314

natives must be evaluated before a counterexample is produced.

Six of the measured periods can easily be shown to be optimal! For a plant with a batch load of 1 this can be readily checked by hand by moving a single batch through the plant and measuring the total duration of the critical branches of the path. In the earlier experiments [5] (which use slightly different plant process durations), we thought that we had made a mistake when we measured the same period for most initialization loads of the plant. 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` (147 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 than $33 + 147 + 26 = 206$ time units. This makes the schedules for loads 2–6 optimal as well.

The schedule for the load of seven batches is almost certainly optimal. It was found by imposing a very tight `maxtime` upper-bound for the production of ten batches, after a more relaxed upper-bound had already produced a period of 346 time units (cf., UPPAAL results below). Further tightening of the upper-bound lead to loss of counterexamples, which means that with a load of seven batches, ten batches cannot be produced within 3320 time units. By itself, this does not prove that there is no shorter period, however, as this might require an initial behaviour that is less productive than that of the schedule that we found.[5]

It must be concluded that the plant can be scheduled in the overall optimal time of 2060 s for all loads, except for the extreme loads of 1 and 7. Because of our analysis above, these are not only time optimal schedules, but also resource optimal ones, in the sense that the (expensive) heating and distillation equipment `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 Optimization with Uppaal

4.1 The UPPAAL *model*

The translation from the plant and control models into a UPPAAL model is quite straightforward, and in many points similar to the philosophy of the Promela model. The main differences between the models are the representation of time, naturally, as this is a built-in feature of UPPAAL, and the scheduling of the concurrent processes. In the Promela model the latter is restricted by fairness conditions and the use of the `cycle` flag variable. The UPPAAL model enforces this by another, more explicit restriction that is explained below.

The UPPAAL model is a parallel composition of a plant automaton and a control automaton. The plant automaton is given as a parallel composition of 12 sub-automata, representing the basic plant processes. Each of these plant automata is equipped with a clock that measures the duration of the process after it has started. As an example, the UPPAAL model for the transport of concentrated salt solution from container B1 to container B3 is given in Fig. 12. Location S0 represents the passive control state of the process. As soon as the control process has opened and closed the appropriate valves, etc., the process starts. This immediate activation is achieved by a synchronization on the *urgent* channel *urgon* (an UPPAAL feature: a channel on which enabled synchronizations cannot be delayed. After the time of the duration of the process has passed, the container volumes are updated depending on their previous values. After a process finishes the control program is activated twice (by synchronization on the channel *compute*), for reasons given below.

As with Promela, the execution of the control automaton is modelled based on the assumption that the execution of the PLC control program is instantaneous. Technically, this can be implemented using the *committed location* feature of UPPAAL. Once reached, committed locations have to be left without time delay or interleaving with other transitions. In our model all locations of the control automaton are modelled as committed locations, apart from the first one that represents the waiting state

```
process P13{
clock x12;
state S0, S1{x12<=32}, S2, S3;
commit S2, S3;
init S0;
trans S0 -> S1 {guard Pump1==0, V8==1,
                V9==0, V11==0, Mixer==1;
                sync urgon?;
                assign x12:=0; },
S1 -> S2 {guard x12==32, B1==1, B3==0;
          assign B1:=0, B3:=2; },
S1 -> S2 {guard x12==32, B1==2, B3==0;
          assign B1:=1, B3:=2; },
S1 -> S2 {guard x12==32, B1==1, B3==1;
          assign B1:=0, B3:=3; },
S1 -> S2 {guard x12==32, B1==2, B3==1;
          assign B1:=1, B3:=3; },
S2 -> S3 {sync compute!; },
S3 -> S0 {sync compute!; };
}
```

Fig. 12. The UPPAAL model of transfer between B1 and B3

[5] Our earlier results [5], which used different process durations, showed easily produced optimal schedules for loads 1,2,3,4, and 7, with optimal schedules obtained for loads 5 and 6 only after substantial tightening of upper-bounds.

of the control process. Without restrictions, repeated instantaneous execution of the control program would lead to Zeno-behaviour. In the Promela model such behaviour is prevented by imposing a fairness requirement. These are not available in UPPAAL. In the UPPAAL model execution of control steps is restricted to those points in time when the conditions in the plant change, as these are the only moments when the control process can change also its state. More precisely, it is the case that each change in the state of the plant requires two control program executions: one to finish some process (close valves etc.), and one to start up new ones. In general, starting up new processes could be delayed for some time, but this is not needed. As was already argued for the Promela model, no (time-)optimal schedules are lost by assuming that new processes only start when some other process finishes.

The control automaton itself consists of two sequential parts, again similar to the Promela model. In the first part the activation conditions for processes are evaluated, including a nondeterministic choice of a subset of the processes that can be activated. As in the Promela case, this nondeterministic step is prerequisite for finding optimal schedules. In the second part the control sets the actuator variables for valves, pumps, heater, and mixer.

4.2 Optimal schedules with cost-optimal UPPAAL

Behrmann et al. [4] proposed an extension of UPPAAL that includes concepts from branch and bound algorithms to provide generic support for optimal search strategies in a real-time context. This extension makes it possible to derive optimal solutions for problems that can be modelled in terms of so-called *Uniformly Priced Timed Automata*. In this model a cost function increases with a fixed rate as time elapses or with a specified amount if a transition is taken. This cost function is implemented using a UPPAAL clock variable with special operations, but such that the currently implemented data structures in UPPAAL suffice.

Among the techniques that were adopted in cost-optimal UPPAAL are heuristic search orders. Heuristic search orders are obtained by assigning priorities to symbolic states. The symbolic state is extended with a *priority* field and the model checker selects the next state with the highest priority. To change priorities one has to decorate the transitions with an assignment to the priority field.

In our cost-optimal UPPAAL model of the case study we have defined the `cost` to be identical to the time elapsed, i.e., the cost rate equals 1, and added the priority variable `heur` that is calculated according to the expression `1*bonus+100*depth-cost`. The variable `bonus` is used to reward selecting larger rather than smaller subsets of enabled process start events, as is shown in Fig. 13. This heuristic directs the exploration such that the controller tries first to start all permissible plant processes. The bonus is made extra rewarding for the selection of the evaporation process, which should be in (almost) continuous use for an optimal exploitation of the plant resources.

The variable `depth` is used to reward a depth-first over breadth-first process scheduling policy by rewarding the starting of enabled plant processes substantially better (by a factor 1000) than their delay (by selecting later transitions of already active processes). This is done as good scheduling solutions are likely have many concurrently active plant processes. Transition `S0 -> S1` in Fig. 14 models the beginning of the process `P13`. We increase the priority if this transition is taken to reward the start of `P13`. Transitions that start other processes and transitions of the controller that enable processes to start are rewarded similarly.

Using this heuristic we could produce optimal schedules for the batch plant without resorting to an initial upper bound `maxtime` on time, as was done using SPIN. The heuristic has a similar effect on the reduction of the search space as the `cuts` parameter in the SPIN case. For the experiments we asked UPPAAL to produce schedules for the same initial load and the same number of batches as SPIN did. Table 3 gives the results of these experiments. For loads 1–6 with the heuristic UPPAAL found an initial solution that has the same period and same simtime as the best solution found by SPIN. The solution for load 7 converges to a schedule with period 346. For the

```
A1 -> A2{assign bonus:=bonus+(T11==1?1:0);},
A1 -> A2{guard T11==1; assign T11:=0;},
A2 -> A3{assign bonus:=bonus+(T12==1 ?1:0);},
A2 -> A3{guard T12==1; assign T12:=0;},
```

Fig. 13. Rewarding the selection of enabled processes (selection variable is not set to 0, if it was 1 before)

```
process P13{
clock x12;
state S0, S1{x12<=32}, S2, S3;
commit S2, S3;
init S0;
trans S0 -> S1{guard Pump1==0, V8==1,
               V9==0, V11==0, Mixer==1;
        sync urgon?;
        assign x12:=0, depth:=depth+1; },
S1 -> S2{guard x12==32, B1==1, B3==0;
         assign B1:=0, B3:=2; },
S1 -> S2{guard x12==32, B1==2, B3==0;
         assign B1:=1, B3:=2; },
S1 -> S2{guard x12==32, B1==1, B3==1;
         assign B1:=0, B3:=3; },
S1 -> S2{guard x12==32, B1==2, B3==1;
         assign B1:=1, B3:=3; },
S2 -> S3{sync compute!; },
S3 -> S0{sync compute!; };
}
```

Fig. 14. The depth is increased if a process starts; with depth the heuristic gives preference to depth-first search

Table 3. This table shows the first solution found by cost-optimal UPPAAL, either with heuristic search or depth-first

		heuristic search		depth first search	
load	batches	simtime	states	simtime	states
1	10	3476	14 063	4178	17 547
2	25	4968	35 952	11 048	45 781
3	25	4732	34 584	10 992	45 385
4	25	4774	35 344	11 288	46 181
5	20	3947	28 808	9294	37 716
6	20	4120	29 568	9294	37 716
7	10	3320	14 377	4418	17 957

same set of problems depth-first search finds worse solutions and explores more states. Other experiments show that depth-first search cannot find a solution at all if we set a reasonable upper-bound `maxtime` on the time, as we did for SPIN.

Cost-optimal UPPAAL offers the option to start backtracking as soon as it finds an initial solution. It uses the cost of this solution to prune states that are more expensive. The bound on the cost is lowered each time it finds a better solution. UPPAAL was not able to find better schedules within reasonable CPU-time and memory for initial loads 1–6. For load 7 it was able to find a schedule with simtime 3288. Backtracking yielded a schedule that produces the last batch within 314 time units; this is the period SPIN found for its best schedules. To do so UPPAAL explores 74 240 states, which takes 16.5 s on a Pentium III 500 MHz.

5 Conclusion

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 in our previous work [13], and on the analysis of formal approaches to PLCs [12].

It is interesting to see that we succeeded in dealing with this real-time embedded system using standard Promela and 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 is a consequence of the tolerance of the plant processes for much slower reaction times than those realized 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 (the average distance in time between events in a trace being close to 20 time units).

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 analogue 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. On the basis of our modified model we could find optimal schedules in an interactive mode: up to five runs with different parameter values, each taking between seconds and a few minutes of user time on a SUN Enterprise E3500-server (most in the seconds range). This is certainly due to the particular characteristics of the given plant, with its very critical heating process. In addition, 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. Given the small numbers of explored states it might seem that the gain by using variable time advance over the explicit addition of a clock variable to the model (as in DTSpin) is immaterial (going from $O(10^3)$ states to $O(10^{4-5})$. One should keep in mind, however, that the number of atomic steps required to produce a counterexample would also grow by a factor 20, taking it to $O(10^{6-7})$, which most likely would interfere with the use of the model-checker in an interactive mode to find (almost) optimal schedules quickly.

Another approach to the optimal scheduling for the VHS case study 1 is reported by Niebert and Yovine [14]. 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 also include the production of the initial loads in their schedules, which we just assume to be present. The more general findings seem to be consistent with ours, however. OpenKronos could be used successfully to produce optimal schedules for loads of up to three batches before falling victim to the state explosion problem. The symbolic model checker SMI produced results for six batches

and more, with a computation time of approximately 17 min per batch.

Since our initial experiments with Promela and SPIN an implementation of cost-optimal UPPAAL has become available. We have used this tool to redo the optimization part of the case study and compared it to our SPIN results. The results obtained with this variant of UPPAAL did confirm correctness of our results. Compared to SPIN, cost-optimal UPPAAL certainly offers a more convenient interface for handling guided state-space explorations. For SPIN the latter can only be done indirectly by repeated verification runs with different parameter settings under control of the user. In cost-optimal UPPAAL the user can control everything directly by defining heuristic cost functions, so that fewer multiple runs are required, although some additional experimentation is needed to fine-tune the heuristic used.

On the basis of the limited experience gained by this case study, the situation is best characterized as follows. Given the current situation, cost-optimal UPPAAL is probably the more convenient tool to search for optimal schedules for this kind of application if an UPPAAL model is available or can be developed. In view of the very reasonable performance of SPIN using the techniques described in this paper, however, developing a UPPAAL model may not pay off if a good Promela model is available, as for this case study. In our case such a model was in fact produced because it proved much easier to develop a Promela model for the (time-abstracted) verification of the plant, then to do so with UPPAAL. Given this situation it can be concluded that it would be very interesting to look also into the possible extension of SPIN with cost-guided state space exploration features. This could prove to be a powerful combination for the application of SPIN to optimization problems. Recent work by Edelkamp et al. [6] combining SPIN with heuristic directed search algorithms also shows the viability of such a combination.

References

1. DTSpin homepage.: `http://www.win.tue.nl/~dragan/DTSpin.html`
2. VHS: sources of case study 1.: `http://www.cs.kun.nl/~mader/vhs/cs1.html`
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor Comput Sci (138):183–335, 1994
4. Behrmann, G., Fehnker, A., Hune, T.S., Larsen, K.G., Petterson, P., Romijn, J.: Efficient guiding towards cost-optimality in UPPAAL. In: 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), 2001
5. Brinksma, E., Mader, A: Verification and optimization of a PLC control schedule. In: Proc. SPIN2000, Lecture Notes in Computer Science, vol. 1885. Springer, Berlin Heidelberg New York, 2000
6. Edelkamp, S., Lafuente, A.L., Leue, L.: Directed explicit model cheking with HSF-SPIN. In: Model checking software, Lecture Notes in Computer Science, vol. 2057. Springer, Berlin Heidelberg New York, 2001
7. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Hytech: a model checker for hybrid systems. Software Tools Technol Transfer (1):110–123, 1997
8. Holzmann, G.J.: The model cheker SPIN. IEEE Trans Software Eng 23(5):279–295, 1997
9. International Electrotechnical Commission: IEC International Standard 1131-3, Programmable Controllers, Part 3, Programming Languages, 1993
10. Kowalewski, S.: Description of case study CS1 "experimental batch plant". `http://www-verimag.imag.fr/VHS/main.html, July 1998`
11. Larsen, K.G., Petterson, P., Yi, W.: Uppaal in a nutshell. Software Tools Technol Transfer (1):134–153, 1997
12. Mader, A.: A classification of PLC models and applications. In: Boel, R., Stremersch, G. (eds) Discrete event systems – analysis and control. Kluwer, Boston, Mass., USA, 2000. Proceedings of WODES2000
13. Mader, A., Brinksma, E., Wupper, H., Bauer, N.: Design of a PLC control program for a batch plant – VHS case study 1. Eur J Control 7:416–439, 2001
14. Niebert, P., Yovine, S.: Computing optimal operation schemes for multi batch operation of chemical plants. VHS deliverable, May 1999. `http://www-verimag.imag.fr/VHS/main.html`
15. Shedler, G.S.: Regenerative stochastic simulation. Academic, New York, 1993
16. Vaandrager, F.W., van Schuppen, J.H.: In: Hybrid systems: computation and control, Lecture Notes in Computer Science, vol. 1569. Springer, Berlin Heidelberg New York, 1999
17. Yovine, S.: Kronos: a verification tool for real-time systems. Software Tools Technol Transfer (1):123–134, 1997