European Journal of Control, 7(4):400-415, 2001. Special Issue: Verification of Hybrid Systems.

Verifying Untimed and Timed Aspects of the Experimental Batch Plant

Ralf Huuck Ben Lukoschus
Institute of Computer Science and Applied Mathematics, University of Kiel,
Preußerstr. 1–9, 24105 Kiel, Germany. {rhu|bls}@informatik.uni-kiel.de

Yassine Lakhnech
Verimag, Centre Equation, 2 av. de Vignate, 38610 Gières, France. lakhnech@imag.fr

Abstract

We thoroughly examine the experimental batch plant in its two major operation modes: a normal operation mode and a failure operation mode. In order to do so, we use discrete condition/event system as well as timed automata for the specification and the model checking tools SMV, Kronos and HyTech for verification.

Keywords verification, hybrid systems, model checking, controller synthesis.

1 Introduction

Safety critical systems have always been one of the main motivations for applying formal methods. Major reasons are that in this field a failure might lead to significant financial deficits, pollution of the environment or losses of life. Within the recent years a number of modeling frameworks came up as well as various tools supporting the specification and verification of these systems.

This contribution targets to present different formal verification approaches on a benchmark case study of a batch plant as proposed in [18]. Briefly, this plant works as follows. Two buffer tanks supply water and a highly concentrated salt solution. These liquids are mixed to get a more diluted solution, which is then evaporated using a tank which is equipped with a heating device. During evaporation, the steam is cooled down in a condenser and captured in another tank. The heating process is stopped when the original high concentration has been reached. Both the resulting high concentrated solution and the captured steam are cooled down and pumped back into the respective storage tanks. The complete production and recycling process is automated and split into small steps, for each of which there exists a program controlling the physical devices of the plant, e.g., valves, pumps, and the heating and cooling devices.

In this article we focus on applying formal techniques to verify untimed as well as timed properties to analyze the safety of this plant. In particular, we distinguish two main operation modes.

The first one is the normal operation. For this mode we formally model the environment of the system as well as the control programs which are given as sequential function charts (SFC) [17]. Since we are interested in general safety properties of the control programs and do not need to consider quantitative time we choose *discrete condition/event systems* (DCES) [25] as a modeling framework. Originated from process control theory this model gained increasing interest in the recent years for the specification of control systems like the experimental batch plant examined here. Since discrete condition/event systems are finite-state, *model checking* [24, 8] can be applied. We use the tool SMV (Symbolic Model Verifier) [21, 22] to perform this verification. Therefore, we first translate a set of given DCESs into an SMV input file, formulate each property as a CTL (Computational Tree Logic) formula [7] and then verify this using SMV.

In a second part we examine the plant behavior in case of a malfunction of the condenser as described in [18]. For this situation hard timing constraints have to be met and, hence, for a formal analysis of this problem we need a timed model. Here we give a specification of the evaporator system using the model of *timed automata* [2] which is widely used in the community of computer scientists. Similar to the first case, we specify the environment—this time incorporating timing behavior—as well as a meaningful controller. For verification we use the model checking tools Kronos [23] and HyTech [12, 11] which allow reasoning about quantitative time.

The remainder of this article is organized as follows: In the first part, Section 2, we consider the untimed case of normal operation. After presenting the modeling framework and the properties we want to check, a specification of the whole system is given as a discrete condition/event system in Section 2.4. Subsequently, the verification is described in detail in Section 2.5. The next part, Section 3, deals with the timed case of failure handling. We specify the system using timed automata in Section 3.1 and present a verification with Kronos in Section 3.2 as well as with HyTech in Section 3.3. Finally, in Section 4 we draw some conclusions.

2 Normal Operation

The goal of this section is to verify safety requirements on the 13 SFC programs given in [18] which control the normal operation of the plant. These requirements forbid certain behaviors that could damage the plant, like switching on the heating device in an empty tank. For verification we will use the tool SMV (Symbolic Model Verifier) [21, 22].

2.1 Modeling Framework

In order to check the requirements formally, we need a modeling framework which allows formal verification. In this framework, not only the control programs are modeled, but also the physical part of the plant which is controlled by the program. Our approach uses *discrete condition/event systems* (DCESs) for both the physical and the control part. *Condition/event systems* (CESs) and their discrete version are introduced in [25] as a modeling paradigm for discrete event systems (DESs) and have gained increasing interest from the DES and hybrid system community (cf. [10], [15], [9]). The main motivation for developing CESs is to provide a framework for modeling DESs based on block diagrams and signal flows as is standard practice in system theory.

A DCES allows to model a system as a relation between its input and output *signals*. A signal can be understood as a function mapping points in time to a finite alphabet of symbols representing time-dependent information. The particularity of DCESs is that they distinguish between two sorts of signals which correspond to two kinds of interactions between interconnected systems, namely *enabling/disabling* of transitions and, on the other hand, *forcing* of transitions. The first kind of interaction is modeled by *conditions* which can be seen as state variables. *Condition signals* are piecewise constant functions and are used to represent time-dependent system states by condition symbols like "on", "off", or "waiting". Enabling of transitions in a DCES may depend on the current condition input symbol. Forcing is implemented by *events*. *Event signals* map points in time to a special *null symbol* **0** except for some discrete points, in which so-called *proper events* occur. Proper events represent instantaneous actions like "start", "stop", or "reset". An enabled transition which is labeled by an event symbol e has to be taken as soon as the event e occurs.

Formally, a DCES is described by a 9-tuple $(U, V, \mathcal{Q}, Y, Z, f, g, h, q_0)$, where U (resp. Y) is the alphabet of input (resp. output) condition symbols, and V (resp. Z) is the alphabet of input (resp. output) event symbols. $\mathcal{Q}$ is the finite set of system states, q_0 is the initial state. The system behavior is determined by the state transition function f, the condition output function g, and the event output function h: if the system is in the discrete state $q \in \mathcal{Q}$, the current condition input symbol is $u \in U$, and the current event input symbol is $v \in V$, then the next discrete state of the system is some $q' \in f(q, u, v)$ (nondeterministic choice), and the condition symbol $g(x, u)$ and the event symbol $h(q, q', v)$ are sent to the outputs of the DCES.

A DCES can have multiple inputs and outputs; if, e.g., there are two independent event inputs with alphabets V_1 and V_2, we can simply use the Cartesian product $V = V_1 \times V_2$ in the framework above.

For more details on CESs and DCESs the reader is referred to [25].

2.2 Assumptions

We make the following assumptions about the plant operation, both of which are fulfilled during the normal operation of the plant:

1. Initially, the tanks are empty, all valves are closed, both pumps are off, and the heater is off.

2. At most one control program is run at a time.

The second assumption allows us to do compositional reasoning, i.e., when checking one program's properties, we do not have to consider the actions of other programs. This has a significant impact on reducing complexity and thus increasing the effectiveness of the verification.

2.3 The Properties

We want to check some safety properties of the control programs. The following ones have to be checked for each program:

- Whenever a control program terminates, all valves are closed.

- Whenever a control program terminates, both pumps are off.

- Whenever a control program terminates, the heater is off.

If these three properties are valid, we are sure that the initial plant state "all valves closed, pumps off, heater off" (called a *safe state* in the following) also holds after the termination of a control program. Together with the assumption above that two programs never run simultaneously, we know that during normal operation of the plant the initial state at the start of a program is always a safe state. As a consequence, the verification of a program's property does not have to consider the actions other programs might have done before or do simultaneously.

However, the states of the tanks (i.e., the water levels) are *not* preserved by the programs. We have to take this into account when verifying the other properties.

To prevent the heating unit and tank 5 from damage by overheating we have the following property:

- Whenever the heater in tank 5 is turned on, the water level of tank 5 is high enough.

We demand that during the heating process steam can leave tank 5 into the condenser only, i.e., all in- and outflow valves of tank 5 are closed:

- Whenever the heater in tank 5 is working, none of the valves 12, 15, and 16 are open.

Furthermore, the cooling device in the condenser must be provided with cold water through valve 13 during the heating process to prevent a dangerous pressure inside the condenser:

- Whenever the heater in tank 5 is working, valve 13 is open.

The regular pressure of the cooling water supply is not sufficient if all 3 cooling units operate at the same time. The following property prevents this undesired situation:

- At most two cooling units are active simultaneously.

Although the pumps are equipped with pressure limit switches, we demand:

- Pumps are not pumping against closed valves.

We do not want an uncontrolled flow of water through a tank:

- Each tank's input and output valves are not open simultaneously.

Now we need a formal model for the plant in which these properties can be checked.

2.4 The Plant Model

We build a discrete, untimed model of the plant, using as modeling paradigm discrete condition/event systems (DCESs) as described in [25]. Based on the plant description in [18], our model consists of two parts. The first part contains the physical devices of the plant, like valves, pumps, and tanks. The second part introduces one DCES for each of the control programs, which are given as sequential function charts (SFCs) in the description.

Communication between the physical devices and the control programs is modeled by conditions providing the device states and by events sent by the programs to control the devices. We associate a name with each connection, making it easier to identify certain conditions and events.

2.4.1 The Physical Devices

We model the following physical devices of the plant:

- The 29 valves. These are either closed or open, and discrete events can be used to force them to change their state.

- The 2 pumps. They are either not pumping or pumping, and discrete events can force them to change their state.

- The 7 tanks. Each of these models the quantity of liquid in a tank. The water level, which depends on the states of the valves connected to the tank, is given as one of two (tank 5: three) discrete values.

- The heating unit. It is either off or on, and discrete events are used to change its state.

Figure 1 shows a block diagram of the interconnections of all physical devices. Condition connections are marked with , and event connections with . The input event connections which can be accessed by the control programs are at the left and right hand side of the diagram.

Since we do not model different concentrations or temperatures of the salt solutions or any pressure differences in pipes or tanks, and since we assume that the condenser is always working properly, the following devices of the plant are not modeled: The mixing unit in tank 3, the condenser, the cooling units, and all kinds of sensors except the level sensors (pressure, concentration, temperature).

We only present a selection of all 39 DCESs here. The states and transitions of the DCES, which define its input/output behavior, are shown in a transition diagram.

The valves. For $i \in \{1, \ldots, 29\}$, valve i is modeled by a DCES $Valve_i = (U, V, \mathcal{Q}, Y, Z, f, g, h, q_0)$, with $V = \{open, close\}$ (with associated name $\mathsf{Cmd\text{-}V}_i$), $Y = \{open, closed\}$ (named $\mathsf{Status\text{-}V}_i$), and $\mathcal{Q} = \{q_0, q_1\}$. There are no condition inputs or event outputs, and therefore, U and Z are chosen in a trivial way, i.e., U some singleton and $Z = \emptyset$. The functions and f, g, and h are defined as follows by the transition diagram of $Valve_i$ in Figure 2:

$$f(q, u, v) = \begin{cases} \{q_1\}, \text{ if } q = q_0 \wedge v = open \\ \{q_0\}, \text{ if } q = q_1 \wedge v = close \\ \{q\}\ , \text{ otherwise} \end{cases},$$

$$g(q, u) = \begin{cases} closed, \text{ if } q = q_0 \\ open\ \ , \text{ if } q = q_1 \end{cases}, \text{ and}$$

$$h(q, q', v) = \mathbf{0}.$$

The condition symbol drawn inside a state is sent to the environment via the condition output whenever control resides in that state. In $Valve_i$ the symbol *closed*, resp. *open*, is sent to the condition output named $\mathsf{Status\text{-}V}_i$ when the DCES is in state q_0, resp. q_1, leading to the definition of g above.

The arrows between the states define the transition function and the event output function. A state transition is taken if the inputs fulfill the symbol list left of the '/' on the transition labeling, and when this transition is taken, the events right of '/' are sent as outputs. In $Valve_i$ the state changes from q_0 to q_1 when *open* appears as an input event for $\mathsf{Cmd\text{-}V}_i$, and from q_1 to q_0 if $\mathsf{Cmd\text{-}V}_i$: *close* occurs, and no output events are sent in either case, leading to the definitions of f and h above.

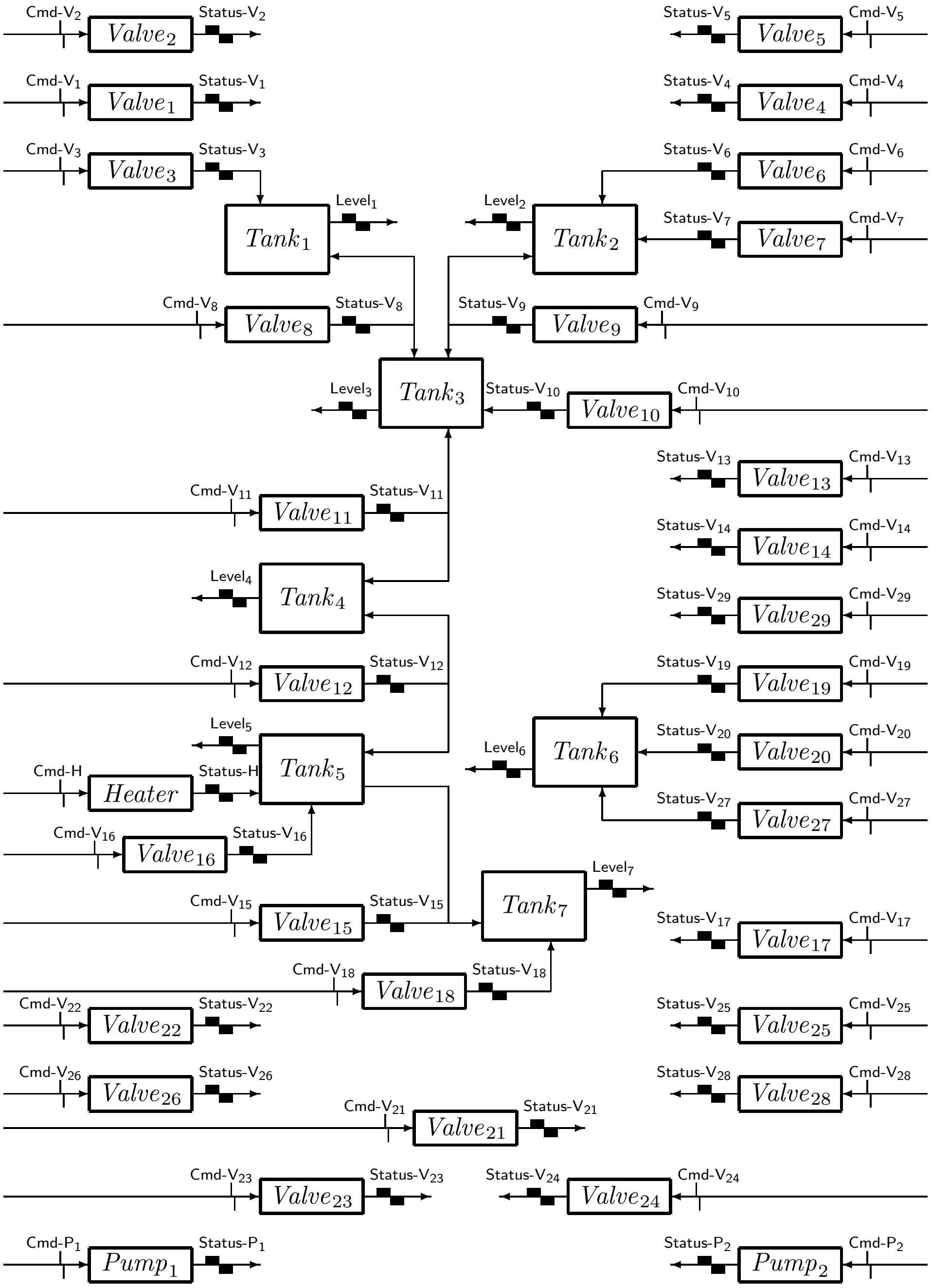

Figure 1: Block diagram of the physical devices modeled as DCESs.

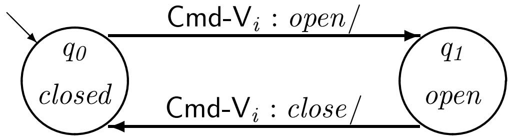

Figure 2: Transition diagram of $Valve_i$.

If a combination of input conditions and events is not covered by any transition labeling, the state of the DCES does not change, i.e., we assume implicit loop transitions with no event outputs for these combinations.

The functionality of $Valve_i$ can be summarized as follows: Initially, the valve is closed. It can be opened by sending the Cmd-V$_i$: *open* event, and closed by sending the Cmd-V$_i$: *close* event. The current state of the valve can be observed as the condition Status-V$_i$ which is either *closed* or *open*. All 29 valves behave in this way.

The pumps. The two pumps are modeled by the DCESs $Pump_1$ and $Pump_2$. Both work similar to the valves. Initially, they are off, and they can be controlled by the events Cmd-P$_1$: *start* and Cmd-P$_1$: *stop*, resp., Cmd-P$_2$: *start* and Cmd-P$_2$: *stop*. Their status is delivered by the conditions Status-P$_1$ and Status-P$_2$, which have the value *off* or *on*.

The tanks. Each DCES model of a tank is used to provide information about the water level in the respective tank of the plant. We use two different abstract levels (*empty* and *full*) in our model, only tank 5 has a third level (*half*). These are output condition symbols of the DCES tank models. Whenever a control program needs to access a water level sensor, it checks the condition output of the respective tank. We do not model the water level sensors themselves.

A change of the water level of a tank depends on the amount of water flowing in and out of the tank. This again depends on pressure differences and the state of the valves. In this model a simple abstraction is chosen: Whenever a valve controlling an input pipe of a tank is open, water may flow into the tank, and the tank's level may change from *empty* to *full*. And whenever a valve controlling an output pipe of a tank is open, water may leave the tank, and the tank's level may change from *full* to *empty*. It is obvious that this is an over-approximation and thus an abstraction of the real plant behavior.

Tank 1 can be filled via valve 3 and emptied via valve 8. Thus, Status-V$_3$ and Status-V$_8$ are condition input names of $Tank_1$. Status-V$_3$: *open* enables the transition from q_0 to q_1 (when taken, this transition yields the condition output change from Level$_1$: *empty* to Level$_1$: *full*), and Status-V$_8$: *open*

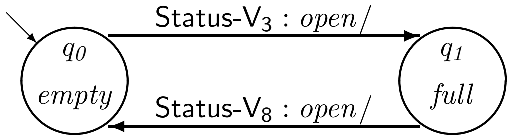

Figure 3: Transition diagram of $Tank_1$.

enables the transition from q_1 to q_0, and if this transition is taken, the condition output changes from Level_1 : *full* to Level_1 : *empty*.

Note that these transitions need not to be taken when they are enabled; in fact they may never be taken, since condition changes cannot force any transitions. This behavior is intended, making this an obvious abstraction of the real plant behavior.

The control programs can read the condition output Level_1 to obtain the current abstracted water level in tank 1.

Tanks 2 to 7 work similarly, only tank 5 has a third state which delivers the condition Level_5 : *half*. We use three values there to have a representation of the water level that suits the structure of control program B5 better than a two-valued representation. Additionally, we model that due to the evaporation process, tank 5 can get empty if the heater is on, and tank 6 can get full at any time, since water may come out of the condenser, even some time after the heating has stopped.

The heating unit. The heater inside tank 5 is modeled by the DCES *Heater*, which is similar to the valves and the pumps. Initially, the heater is off. Heating can be started by sending the Cmd-H : *start* event and stopped by sending the Cmd-H : *stop* event. The current state of the heater can be observed at the Status-H condition output, which can be *off* or *on*.

2.4.2 The Control Programs

The control programs are given in [18] as Sequential Function Charts (SFCs). We implemented these SFCs as DCESs, trying to stay as close to their original structure as we can in our abstract framework. Each numbered step i of the SFC becomes a control state q_i in the DCES. The commands executed in step i are represented in the events generated when entering q_i. Conditions which enable the change from step i to step j are the enabling conditions of the DCES transition from q_i to q_j.

The following parts of the SFCs are not modeled, since we do not model the respective physical devices: User interaction with a display and some confirmation input, concentration sensor inputs, the cooling liquid flow sensor, and temperature sensors.

All DCES control programs are started by an external event $\mathsf{Cmd\text{-}Prg_N}$: *start*, where N is the name of the program. When this event occurs, the control state changes from q_1 to q_2, and the control program interacts with the physical devices until some final control state is reached. In contrast to the SFCs, the DCESs only do one cycle, i.e., they do not return to the initial control state when they have reached some final control state.

We illustrate the DCES models of the control programs by program B5.

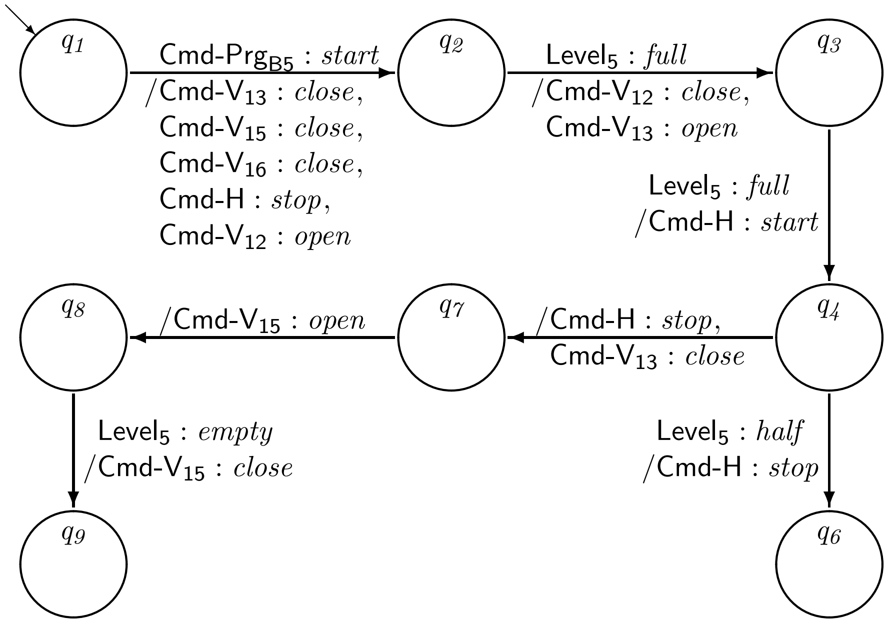

Figure 4: Transition diagram of Prg_{B5}.

Program B5. This program controls the evaporation of the concentrated brine in tank 5 until the desired concentration is reached. First some concentrated brine from tank 4 is filled into tank 5 via valve 12 until tank 5 is full. Then the cooling in the condenser is started by opening valve 13. The heater is switched on, and the evaporation starts. If the desired concentration is reached (not modeled), the heater is switched off, and then tank 5 is drained into tank 7 via valve 15.

If during the evaporation process the water level in tank 5 goes below 8 cm (modeled as $\mathsf{Level_5}$: *half*), the heater is switched off, and we enter state q_6, denoting a failure.

The transition diagram of the DCES Prg_{B5} is shown in Figure 4.

2.5 Verification

For each of the desired properties and each control program involved in the validity of this property we perform the following steps to verify it:

1. Generate the SMV code for the DCESs we need to model check the property.

2. Insert the property into the SMV code using a `SPEC` declaration.

3. Use SMV to verify the property.

The model checking tool SMV [21, 22] can be used to verify finite transition systems. The language for specifying the system allows the use of variables with finite domains and supports structuring with modules. The behavior of the system can be given either as a set of variable assignments or as a transition relation. The properties to be checked have to be provided as CTL (Computational Tree Logic) formulae [7]. Internally, SMV builds the transition relation using BDDs (Binary Decision Diagrams) as a memory-efficient data structure. If a property is not met by the system, a counterexample is produced by SMV.

We illustrate the three steps above in detail with the property "Whenever the heater in tank 5 is turned on, the water level of tank 5 is high enough":

The DCESs we have to inspect are Prg_{B5} (since this is the only control program sending commands to the heater) and $Tank_5$ (because we must take a look at its condition output Level_5). Thus, we build an SMV file (see Figure 5) with one module `PrgB5_module` for the DCES Prg_{B5} and one module `Tank5_module` for the DCES $Tank_5$. The communication via events between the modules is modeled by shared variables (first `VAR` declaration). For technical reasons we need the variable `conditionsymbols` which lists all the possible values for conditions in this system. The third `VAR` declaration instantiates the two modules as `PrgB5` and `Tank5`. The module parameters in these instantiations are used to describe the signal connections between the components; in this case, the local variable `Level5` of `PrgB5` (a condition input) is identified with the local variable `Level5` of `Tank5` (a condition output). The `INIT` declarations state that initially there are no events and that the programs are in their initial state. The `DEFINE` declaration in `Tank5_module` specifies which condition output symbols are delivered to `PrgB5` which reads the variable `Level5`; in other words, the function g of the DCES $Tank_5$ is modeled here. The use of `DEFINE` is possible for all condition outputs and saves some memory during model checking, since no additional variables for the condition outputs have to be introduced.

For the sake of brevity we left out the `TRANS` declarations which define the transitions and the event output behavior of the modules, i.e., which model the DCESs' functions f and h. Technical restrictions make it impossible to use `DEFINE` declarations for events.

```
MODULE main

VAR  StatusV12: {closed,open};  StatusV15: {closed,open};
     StatusV16: {closed,open};  StatusH: {off,on};
     CmdPrgB5: {0,start};

VAR  conditionsymbols: {closed,open,off,on,empty,half,full};

VAR  Tank5: Tank5_module(StatusV12,StatusV15,StatusV16,StatusH);
     PrgB5: PrgB5_module(Tank5.Level5,CmdPrgB5);

INIT  CmdPrgB5=0

SPEC  AG (PrgB5.CmdH=start -> !Tank5.Level5=empty)


MODULE Tank5_module(StatusV12,StatusV15,StatusV16,StatusH)

VAR  state : {q_0,q_1,q_2};

INIT  state in {q_0}

DEFINE  Level5 := case
                    state=q_0: empty;
                    state=q_1: half;
                    state=q_2: full;
                  esac;

TRANS  [...]


MODULE PrgB5_module(Level5,CmdPrgB5)

VAR  state : {q_1,q_2,q_3,q_4,q_6,q_7,q_8,q_9};

VAR  CmdV12: {0,open,close};  CmdV13: {0,open,close};
     CmdV15: {0,open,close};  CmdV16: {0,open,close};
     CmdH: {0,start,stop};

INIT  state in {q_1}

INIT  CmdV12=0 & CmdV13=0 & CmdV15=0 & CmdV16=0 & CmdH=0

TRANS  [...]
```

Figure 5: SMV input file for the parallel composition of Prg_{B5} and $Tank_5$.

We have written a compiler which automatically translates a subset of the graphically defined DCESs into an SMV input file. It also does some consistency checks like ensuring that each output signal is generated by exactly one DCES.

Finally, we formulate the property we want to show as a CTL formula, which states that whenever the heater receives the command to switch on, the status of tank 5 tells that it is not empty:

$$\forall\Box(\mathsf{Cmd\text{-}H} = \mathit{start} \Rightarrow \mathsf{Level}_5 \neq \mathit{empty}).$$

We insert the above formula into the SMV file using the following `SPEC` declaration (note that variable names must be preceded by the module that is writing the variable):

```
SPEC  AG (PrgB5.CmdH=start -> !Tank5.Level5=empty)
```

This results in the SMV input file as shown in Figure 5.

Now we use SMV to verify the property. The output of SMV is:

```
-- specification AG (PrgB5.CmdH = start ->
                          !Tank5.Level5 ...  is true
```

Thus, control program Prg_{B5} satisfies the property.

We could have put more than just Prg_{B5} and Tank_6 into the parallel composition to verify the property, like the valves 12, 15, 16, or the heater. This is not necessary, since these additional systems are not needed. The outputs of these systems are defined as global variables (`StatusV12`, `StatusV15`, `StatusV16`, and `StatusH` in the first `VAR` declaration in Figure 5), and SMV chooses arbitrary values for these variables in each computation step. In other words, we have "chaotic" behavior of the physical devices not contained in the parallel composition, which is a typical test situation for open systems. So any behavior of the parallel composition of Prg_{B5}, Tank_6, and some more systems is also a behavior of the parallel composition of just these two systems.

We are only investigating properties that describe all possible behaviors, i.e., CTL formulae of the form $\forall\Box(\varphi)$. Thus, we can always try to compose a small set of systems, and if the verification fails due to a behavior that is not a behavior of the complete system, we can add more systems to the parallel composition until we succeed.

Now we start to verify if the control programs fulfill our list of properties. In the following we will list for each property which systems have to be composed, and we discuss the verification results.

Whenever a control program terminates, all valves are closed. All control programs open at least one valve, so we compose each control program with the valves it sends commands to. We check if the status variable of each valve has the value `closed` when the control program is at one of its terminating states. Since we assume that all valves are closed when a program is started and two programs do not run simultaneously, we do not need to consider valves that are not switched by the program.

The verification with SMV reveals that all programs fulfill the property, with one exception: SMV shows the following error trace when verifying program Prg_{B5} (we omitted some less interesting output):

```
state 1.1:
Valve12.StatusV12 = closed
Valve13.StatusV13 = closed
Valve15.StatusV15 = closed
Valve16.StatusV16 = closed
PrgB5.state = q_1

state 1.2:
CmdPrgB5 = start
Valve12.StatusV12 = open
PrgB5.state = q_2

state 1.3:
Valve12.StatusV12 = closed
Valve13.StatusV13 = open
PrgB5.state = q_3

state 1.4:
Level5 = half
PrgB5.state = q_4

state 1.5:
PrgB5.state = q_6
```

If control program Prg_{B5} enters state q_6 this indicates that during the evaporation process the water level in tank 5 has fallen too low. Therefore the heater is switched off, and the program stops. The error trace above however shows that valve 13, which was opened earlier when entering state q_3, is still open at the final state q_6.

Although this is a violation of our property, it is not that severe, since we can assume that after this error state q_6 is reached, there is some intervention by the user such that other programs will not be started before the plant is reseted to its initial state, and this reset includes the closing of all valves.

Furthermore, a permanent open state of valve 13 does not endanger the plant, since valve 13 only controls the cooling water supply for the condenser. If, e.g., valve 12 would remain open permanently, an overflow of tank 5 might occur.

Another interesting case is the verification of program Prg_{SP1}. This program controls 19 different valves, and thus, the resulting state space of the parallel composition of all these valves and Prg_{SP1} is quite large. As a consequence, the verification took about 23 hours on a Sun Ultra 1 Model 170 system running at 167 MHz and consumed about 340 Megabytes of main memory.

In this special case avoiding this problem is easy. We can split up the task into 19 subtasks, each verifying that one of the valves is closed when Prg_{SP1} terminates. In this case, each parallel composition only involves two systems, Prg_{SP1} and one of the 19 valves, making the state space much smaller. Each of these 19 checks then succeeds within a fraction of a second.

Whenever a control program terminates, the pumps are switched off. There are five control programs which send commands to pump 1: Prg_{B3U}, Prg_{B6S}, Prg_{B7U}, Prg_{SP1}, and Prg_{SP2}. We compose each of these programs with $Pump_1$ and check if the status variable of pump 1 has the value `off` when the control program is at its terminating state. SMV shows that all five programs are correct w.r.t. this property.

Analogously, we check this for program Prg_{B6U}, which is the only program using pump 2. This check also succeeds.

Whenever a control program terminates, the heater is switched off. Only control program Prg_{B5} uses the heater. We compose Prg_{B5} and *Heater* and check if the heater's status variable is `off` when the program is in one of its final states. This is the case.

Now we know that whenever a control program terminates, all valves are closed (with the one exception listed above), and the pumps and the heater are switched off, provided this was also the case when the control program was started. We further assume that two programs are not running simultaneously. This allows us to easily verify the rest of our properties.

Whenever the heater in tank 5 is turned on, the water level of tank 5 is high enough. Only control program Prg_{B5} uses the heater. We compose Prg_{B5} and $Tank_5$ (cf. Figure 5) and check if the tank's water level does not have the `empty` value when the Cmd-H : *start* event occurs. We do not need to include the heater in the composition, since we are only interested in the starting command generated by the control program. SMV shows that Prg_{B5} fulfills this property.

Whenever the heater in tank 5 is working, none of the valves 12, 15, and 16 are open. Again, only control program Prg_{B5} uses the heater. We compose Prg_{B5}, *Heater*, $Valve_{12}$, $Valve_{15}$, and $Valve_{16}$, and check if the status variable of each valve is *closed* if the heater status is *on*. This property holds.

Whenever the heater in tank 5 is working, valve 13 is open. We compose Prg_{B5}, *Heater*, and $Valve_{13}$, and check if the status variable of valve 13 is *open* if the heater status is *on*. This property also holds.

At most two cooling units are active simultaneously. Each of the three cooling units is active when the valve through which the cooling water flows is open. These are the valves 13, 17, and 29. Since there is no control program which sends commands to all three valves and since two programs never run simultaneously, this property is trivially fulfilled.

Pumps are not pumping against closed valves. We show that whenever a pump is working, the water flowing through the pump can leave the system via valve 26 or valve 28, or is pumped into tank 1 or tank 2. For pump 1, this leads to four possible paths:

1. From pump 1 through valves 22 and 26.
2. From pump 1 through valves 22, 1, and 3 into tank 1.
3. From pump 1 through valves 22, 1, 2, 4, and 6 into tank 2.
4. From pump 1 through valves 22, 1, 2, 4, 5, and 28.

We formulate the property that along at least one of these paths all valves are open when pump 1 is working. Pump 1 is used by the control programs Prg_{B3U}, Prg_{B6S}, Prg_{B7U}, Prg_{SP1}, and Prg_{SP2}. We compose each of these programs with all the valves mentioned above. SMV shows that the property is fulfilled by all programs.

For pump 2, we analogously have the following four paths:

1. From pump 2 through valves 25 and 28.
2. From pump 2 through valves 25, 5, and 6 into tank 2.
3. From pump 2 through valves 25, 5, 4, 2, and 3 into tank 1.
4. From pump 2 through valves 25, 5, 4, 2, 1, and 26.

Pump 2 is only used by control program Prg_{B6U}. SMV shows that this program also keeps at least one path of valves open when pump 2 is working.

Each tank's input and output valves are not open simultaneously. This property ensures that we have a controlled flow of liquids in the plant, i.e., no program lets water flow directly through a tank without storing it first in that tank. For all tanks, we check this property on each program that is controlling at least one of the tank's input valves and at least one of the tank's output valves.

For tank 1, we check the programs Prg_{B3U} and Prg_{SP1}, for tank 2 the programs Prg_{B2} and Prg_{SP1}, for tank 3 the programs Prg_{B3}, Prg_{B3K}, Prg_{B3U}, and Prg_{SP1}, for tank 5 the program Prg_{B5}, and for tank 6 the program Prg_{SP2}. For tanks 4 and 7 nothing has to be shown, since there is no program that is controlling both the input and output valves of one of these tanks.

All checks succeed, with one exception: When checking $Tank_3$ with Prg_{SP1}, a trace is generated by SMV showing that valve 9 is opened in one step, and valve 10 is opened in the following step. This violates our property, since water is now flowing through tank 3. Since program Prg_{SP1} is used for maintenance purposes only and is not a part of the normal production cycle, this is not considered to be a severe error.

3 Failure Handling

In this section, we are not interested in handling the normal operation, but in the system's reaction to certain malfunctions which may endanger the safety of the plant. Figure 6 shows the part of the plant we are going to examine. It mainly consists of four physical devices (note that their names differ from those of the original plant): The two tanks (tank 1, tank 2) in which liquids and steam can be kept, the condenser which is able to cool steam down to water, and the heating device (heater) for increasing the temperature of the liquid inside the first tank. Moreover, there are pipes between the devices which can be opened or closed using valves.

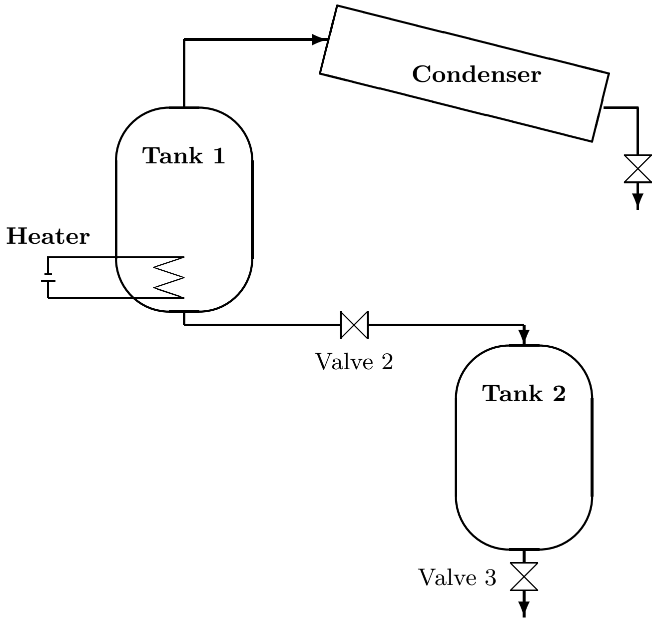

Figure 6: Diagram of the evaporator system.

We consider the following scenario: During the evaporation of the solution in tank 1, the condenser malfunctions. Hence, the steam cannot be cooled, and the pressure inside the condenser starts to rise. Now the heater in tank 1 has to be switched off to prevent further production of steam, since exceeding a certain pressure limit may cause the condenser to explode.

After switching off the heater we have to care about another problem: If the solution in tank 1 cools down below a certain temperature, it may become solid and cannot be drained into tank 2. Therefore, valve 2 must be opened early enough to let the solution be drained into tank 2, where a post-processing step prevents the solution from becoming solid. However, before valve 2 can be opened, we must ensure that tank 2 is empty; otherwise an overflow occurs in tank 2. So we might have to open valve 3 first to drain tank 2.

Our task is to model a controller which can cope with the problems discussed above. The controller, which can influence the system by switching off the heater and using valves 2 and 3 to drain both tanks, must satisfy the following safety properties:

1. If a cooling malfunction in the condenser occurs, the heater must be switched off early enough such that the condenser does not explode.

2. If the heater is switched off, tank 1 is drained into tank 2 before the solution is in danger of becoming solid, this means, the heater must not be switched off too early.

3. If valve 2 is opened to drain tank 1 into tank 2, tank 2 has to be empty.

Later on we will give formalized versions of these properties for each model we use.

The following timing parameters are given by the physical characteristics of the batch plant:

- If the condenser malfunctions and the production of steam in tank 1 continues, the condenser may explode after more than 8 time units.

- If the heater in tank 1 is switched off, it takes exactly 5 time units until the production of steam stops.

- If the production of steam stops in tank 1, the solution may become solid after more than 4 time units.

- It takes between 0 and 10 time units to empty tank 2.

The draining of tank 1 is supposed to be quick enough, so no timing constraint is given here.

3.1 Specification as Timed Automata

Timed automata (TAs) [2] are finite state automata extended by real-valued variables called *clocks*. Semantically, timed automata allow two kinds of transitions, namely *discrete* and *time-pass transitions*. The discrete transitions are specified by the edges between the control locations and are guarded by predicates over the variables. A discrete transition can only be taken when the values of the variables satisfy its guard. Moreover, when a discrete transition is taken, some of the variables may be reset. While control resides at a control location of the automaton time can pass and the values of the variables increase by the amount of time that passed as long as an *invariant,* a boolean expression over the clocks, associated to this location holds. Timed automata are composed using the parallel composition operator, where they have to synchronize on common events.

The verification problem for timed automata has been intensively investigated and many interesting logics have been identified for which this problem is decidable [1, 3, 5, 6, 19]. Also the synthesis problem for timed automata is decidable [20].

Here we give a specification of the evaporator system using timed automata. This specification is the basic model for the subsequent "implemented" specifications in the input languages for the tools Kronos [23] and HyTech [12, 11]. First, we decompose the evaporator system into the following less complex subsystems: The heater, tank 1, tank 2, the condenser, its pressure, and the controller. Next, we give the graphical representation for each component. Note, throughout the description we use the notions "send" and "receive" to illustrate the communication by indicating a direction, but in fact there is only a synchronization on common events for timed automata.

The Heater. The heater behaves as follows: After receiving the event *switch_off* from the controller it resets its clock h and remains in the location *waiting* for 5 time units. Afterwards, it indicates that no more steam emerges by sending *heater_off*. The combination of the invariant $h \leq 5$ and the guard $h = 5$ of the transition forces it to be taken at this particular moment 5.

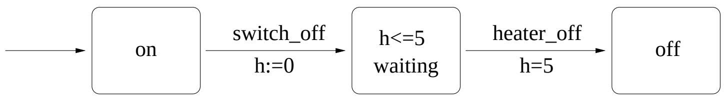

Figure 7: Heater.

Tank 1. We assume tank 1 is full (at any level). If the event *drain1* from the controller occurs, tank 1 will be empty instantly. However, if the *heater_off* event occurs, this means the temperature has fallen below some certain level, the clock *t1* is reseted and if it rises over 4 without obtaining the *drain1* event the solution in tank 1 may become solid and we reach the location *solid*. Note that there is no invariant "$t1 \leq 4$" in the *danger* location, since the transition to *solid* may be taken after 4 time units, but is not forced.

Tank 2. Initially, tank 2 is filled. Then, there is a malfunction within the system and an emergency draining has to be initiated to guarantee a safe shutdown of the system. This is invoked by *emer_drain*. But if tank 2 isn't empty yet and tank 1 is already being emptied there might be an overflow in tank 2. Since it takes up to 10 time units to drain tank 2, this is expressed in the invariant of the *draining* location, every *drain1* event within this period might lead to an overflow situation. However, after 10 time units tank 2 will reach the *empty* location which it indicates by the event *empty2*.

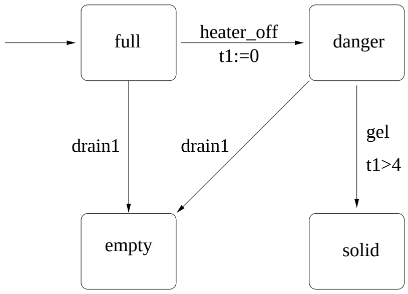

Figure 8: Tank 1.

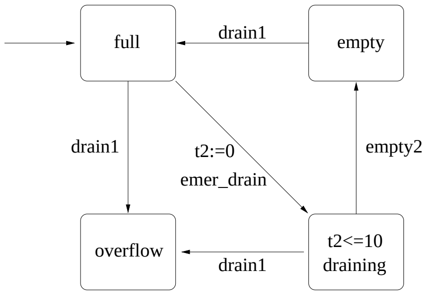

Figure 9: Tank 2.

The Condenser. For our purpose we take a very simple view on the condenser. It is working or not. Initially, the condenser is well functioning but it may suddenly break down, which is indicated by the *mal* (=malfunction) event.

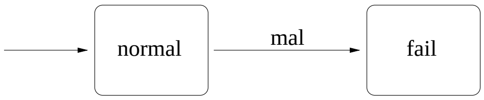

Figure 10: Condenser.

Pressure Within the Condenser. Since we like to keep our model of the evaporator system as modular as possible we define the condenser and its pressure inside as two distinct subsystems. While the condenser just indicates whether it is broken or not, the pressure indication works at follows: From the initial *normal* location it goes to some *danger* location on the event *mal*. From there it either may reach a state of too high pressure after more than 8 time units or returns to the *normal* state if the *heater_off* event occurs in time. Like in tank 1 there is no invariant in *danger* which forces the transition to *p2high*.

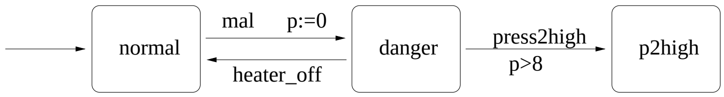

Figure 11: Pressure.

The Controller. The most complex subsystem is of course the controller. When the malfunctioning event *mal* is received from the condenser, the event *emer_drain* is sent to tank 2 within 1 time unit, and an emergency draining is initiated for tank 2. Then the heater is switched off by receiving the event *switch_off* from the controller. This must occur more than 2 time units after *emer_drain* was sent, and within 3 time units after *mal* was received. If the *empty2* event arrives from tank 2 earlier, *switch_off* is sent immediately. Next, we wait for the event *heater_off* to arrive from the heater and *empty2* to arrive from tank 2 (if it has not been received yet). Exactly 4 time units after the *heater_off* event was received from the heater, the *drain1* event is sent to tank 1. Hence, tank 1 is emptied and the system is safe.

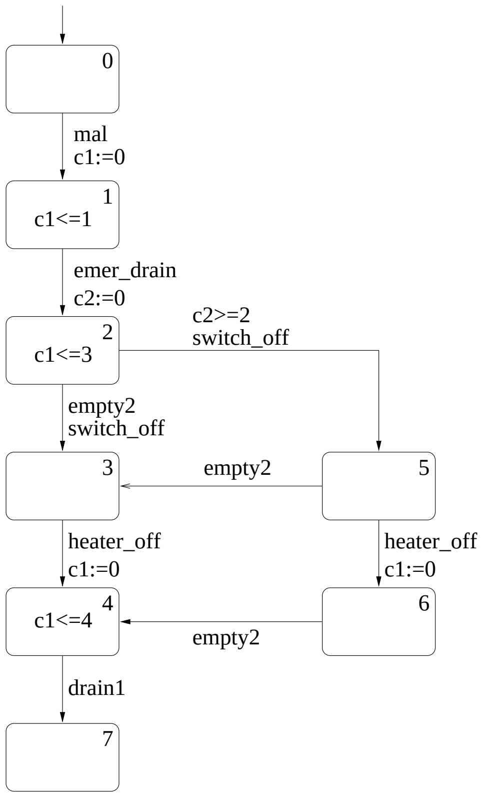

Figure 12: Controller.

3.2 Verification with Kronos

Kronos [23] is a model-checking tool based on the theory of timed-automata and temporal logics. Generally, each automaton of a system can be specified on its own. Moreover, TCTL (Timed Computational Tree Logic) formulae [13]—an extension of CTL for quantitative temporal reasoning over dense time—describing the requirements of the systems can be given. Then, Kronos constructs the resulting product automaton automatically and checks whether the composed system will satisfy these requirements.

Concerning this case study we specify the given components, i.e., the tanks, the heater, the condenser, its pressure, and the controller straightforward from the formal specification as timed automata. In order to give an impression of this we explain the specification of the heater as Kronos input code.

```
#locs 3
#trans 2
#clocks h
#sync  switch_off heater_off


loc: 0
prop: HEATER_ON
invar: true
trans:
 true => switch_off; h:=0; goto 1

loc: 1
prop: HEATER_WAITING
invar: h<=5
trans:
 h=5 => heater_off; ; goto 2

loc: 2
prop: HEATER_OFF
invar: true
trans:
```

The whole component "heater" consists of 3 *locations*, 2 *discrete transitions* and one *clock*, namely h, and the two synchronization events *switch_off* and *heater_off*. The specification derived from the timed automata model is quite straightforward. One has to specify each location by:

1. an identifier for the location,

2. a set of boolean variables that evaluate to *true* whenever control resides at this particular location, called *propositions*,
3. a predicate over the values of the clocks, called *invariant*, that allows control to reside at this location as long as it holds, and
4. a set of outgoing transitions. Each transition consists of a predicate over the clocks, called *guard*, a set of events and a set of assignments resetting clocks to zero.

In order to analyze the heater's process a TCTL formula can be used to examine, e.g., whether along any execution starting from the initial location the heater finally will produce no more steam, i.e., reach the location number 3. This means, the proposition *HEATER_OFF* becomes true. This is expressed in TCTL as

$$init \Rightarrow \forall\Diamond HEATER\_OFF$$

and formalized in Kronos as

`init impl ad HEATER_OFF .`

In the same way we specify all the other mentioned components. The whole system is obtained by parallel composition of its subsystems and can be done automatically by Kronos. For the whole system we check the desired requirements, e.g., we prove

$$\begin{array}{ll} init \Rightarrow \forall\Box & \neg(TANK1\_GEL \\ & \vee TANK2\_OVERFLOW \\ & \vee PRESSURE\_PRESSURE2HIGH) \end{array}$$

holds. In words, starting from the initial states for all executions we are never in a state where the solution in tank 1 gets solid or an overflow of tank 2 occurs or the pressure within the condenser becomes too high.

On the other hand, we prove:

$$\begin{array}{lll} init \Rightarrow & \forall\Box & (CONTROLLER\_LOC1 \\ & \Rightarrow \forall\Diamond & (TANK1\_EMPTY \\ & & \wedge TANK2\_FULL \\ & & \wedge HEATER\_OFF \\ & & \wedge PRESSURE\_NORMAL \\ & & \wedge CONTROLLER\_LOC7)) \; . \end{array}$$

This means whenever we start from the initial states and a malfunction in the system occurs (controller moves to location number 1) the system will reach the safe states. Hence, we always will have a safe shutdown of the system.

The verification of the different properties on a Sun SPARC Ultra I workstation takes less than one second. The reason for this nearly neglectable duration is that the controller dictates a very stringent shutdown sequence. There is not much freedom in terms of non-determinism and, hence, the resulting product automaton will be relatively small. Moreover, each single automaton has as most seven locations, which also contributes to a moderate size of the product.

3.3 Verification with HyTech

In this section we examine the same problem, using a different tool, namely, HyTech [12, 11]. In contrast to Kronos, the model checking tool HyTech allows to analyze linear hybrid systems, which may have continuous variables with different change rates in different locations. Moreover, the tool supports not only the checking of system requirements but also parametric analysis. We will make tactical use of this to synthesize some of the clock thresholds in the controlling process. Controller synthesis is gaining increasing interest [4, 26], and various case studies have been examined using features of HyTech [14] and other tools [16]. However, in contrast to Kronos HyTech does not support the checking of a model against a TCTL formula but allows checking for reachability of defined regions only.

Again, to obtain a specification in HyTech is straightforward from the formal timed automata model. In order to give some impression on HyTech, let us take a look at the specification of the heater as HyTech input code:

```
var h : clock;

automaton heater
initially ON & h=0;
synclabs: switch_off, heater_off;

loc ON: while True wait {}
        when True sync switch_off do {h'=0} goto WAITING;

loc WAITING: while h<=5 wait {}
        when h=5 sync heater_off do {} goto OFF;

loc OFF: while True wait {}
end
```

Generally, the kind of variables used in the formalism, e.g., clocks, are specified globally. Moreover, for each automaton the initial state has to be defined as well as the synchronization labels used. Again, we use *switch_off* and *heater_off* for this purpose. In contrast to Kronos you can directly associate a name with each location. Furthermore, each location has an invariant with an optional rate condition specifying the rate at which the value of some variable progresses. The transitions are specified by a guard, one synchronization event and the successor states for a set of variables as well as the target location.

Analysis can be done by defining regions and then checking whether they are reachable. Briefly, a region determines a set of states by Boolean combinations ("&" is used for conjunction and "|" for disjunction) of location constraints, indicated by `loc`, and linear inequalities over the continuous variables. We define an initial region

```
init_reg := loc[tank1] = FULL & t1=0 &
            loc[tank2] = FULL & t2=0 &
            loc[heater] = ON & h=0 &
            loc[condenser] = NORMAL &
            loc[pressure] = NORMAL & p=0 &
            loc[controller] = LOC0 & c1=0 & c2=0 & aux=0;
```

where all automata are in their initial locations and their clocks are set to zero. Then we use the following HyTech instruction to compute the region *reached* that is reachable from *init_reg*:

```
reached := reach forward from init_reg endreach;
```

Furthermore, we define a final region

```
final_reg := loc[tank1] = SOLID  |
             loc[tank2] =  OVERFLOW |
             loc[pressure] = P2HIGH;
```

subsuming the locations where the solution solidifies, we have an overflow in tank 2 or the pressure becomes too high. Obviously, the final region just describes the states we want to prevent from being reachable from the initial one. And indeed, we can check

$$\mathit{reached} \cap \mathit{final\_reg} = \emptyset,$$

since HyTech evaluates `reached & final_reg` to an empty region. This means, whenever we start from the initial region we will never encounter a situation we like to prevent.

It is always a good idea to check if the system does anything at all. By making small errors in the automata designs, it is possible that the complete system always gets blocked in the initial state, and the controller is never challenged to perform his task. Then the safety result above would be worthless. Therefore, we define a safe region

```
safe_reg := loc[heater] = OFF &
            loc[tank1] = EMPTY &
```

```
loc[tank2] = FULL &
loc[pressure] = NORMAL &
loc[controller]= LOC7;
```

(the region of situations where we have a safe shutdown of the batch plant after a condenser malfunction), and we can show that

$$\mathit{reached} \cap \mathit{safe\_reg} \neq \emptyset$$

holds. This means, from the initial region a desired "good" state is reachable. It cannot be proven in this way that we always end up in a good state, since the malfunction event may not occur at all. However, we do know now that the system is not always blocked in the initial state or a non-safe state.

One of HyTech's strong features is to use parametric analysis in order to, e.g., synthesize thresholds for the controller. Let's take a look at a part of the controller's specification:

```
loc LOC4: while c1<=4  wait {}
      when True sync drain1 do {} goto LOC7;
```

This location is responsible to force the draining of tank 1 within 4 time units after the heater as indicated to be off. However, using parametric analysis we can substitute the threshold 4 by a parameter, e.g., `beta`. Hence, we obtain:

```
loc LOC4: while c1<=beta  wait {}
      when True sync drain1 do {} goto LOC7;
```

Running the analysis again, i.e., checking for reachability of the undesired final region produces the following result:

$$\texttt{beta} > 4.$$

This means, it would be reachable if *c1* is greater than 4. In the same way we can synthesize the other threshold values. As it turns out, the parameters describing the most accurate controller are just the ones we chose beforehand.

The verification times for the presented properties are around one second and even for the parametric analysis they are just slightly higher. The reasons are the same as in the previous section.

4 Conclusions and Future Work

In this paper we presented a comprehensive study of the experimental batch plant. Major requirements were modeled and checked automatically by different tools appropriate to the task to perform.

A discrete, untimed model of the batch plant and its control programs has been used to verify safety properties of the plant. These properties help to ensure that the control programs do not cause damages to the plant's devices. The formal verification reveals that all properties are fulfilled, with two exceptions. In these cases SMV shows the trace of a computation prefix leading to the state where the error occurs. This trace provides a convenient way of finding the error in the control program that causes the violation of the property. The state space of even this relatively small plant is too large prohibiting the construction of a global model on which verification can be performed. The solution we adopted in this work is to decompose the verification task into several verification tasks involving smaller state spaces. This technique is known as *compositional verification.*

The timed model served to construct and verify a controller which prevents further damage of the plant in case of a given failure. We showed how to translate the timed automata specification directly to the input language of the real-time model checking tools Kronos and HyTech, where we verified all major properties concerning the safety of the system.

Further work lies in automated techniques to develop heuristics to choose the DCESs that have to be composed for checking a given property, in methods for reducing time and space consumption (like the splitting of a formula for several verification steps), and in using other levels of abstraction for the models. Moreover, a deeper investigation into the parametric capabilities of HyTech seems to be interesting in order to make improvements in controller synthesis.

Acknowledgments. This work was supported by the European Community in the Esprit Long-Term Research Project 26270 – VHS (Verification of Hybrid Systems). Moreover, the authors like to thank the anonymous referees for valuable suggestions and comments.

References

[1] R. Alur, C. Courcoubetis, and D.L. Dill. Model Checking for Real–Time Systems. In *LICS'90*, pages 414–425, 1990.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] R. Alur, T. Feder, and T.A. Henzinger. The Benefits of Relaxing Punctuality. *Journal of the ACM*, 43:116–146, 1996.

[4] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*, pages 469–474. IFAC, Elsevier, July 1998.

[5] A. Bouajjani and Y. Lakhnech. Temporal logic + timed automata: Expressiveness and decidability. In I. Lee and S.A. Smolka, editors, *CONCUR '95: concurrency theory: 6th International Conference, Philadelphia, PA, USA, August 21–24, 1995*, volume 962 of *LNCS*, pages 531–546. Springer-Verlag, 1995.

[6] A. Bouajjani, Y. Lakhnech, and S. Yovine. Model checking for extended timed temporal logics. In B. Jonsson and J. Parrow, editors, *FTRTFT '96: Formal techniques in real-time and fault-tolerant systems: 4th international symposium, Uppsala, Sweden, September 9–13, 1996*, volume 1135 of *LNCS*, pages 306–326. Springer-Verlag, 1996.

[7] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logics of Programs Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *LNCS*. Springer-Verlag, 1982.

[8] E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symposium of Programming Languages*. ACM Press, 1983.

[9] S. Engell and I. Hoffmann. Modular Hierarchical Models of Hybrid Systems. In *35th IEEE Conf. on Decision and Control*, Kobe, Japan, 1996.

[10] H.-M. Hanisch and M. Rausch. Synthesis of Supervisory Controllers Based on a Novel Representation of Condition/Event Systems. In *IEEE Int. Conf. on Systems, Man., and Cybernetics*, Vancouver, Canada, 1995.

[11] T.A. Henzinger and P.-H. Ho. HyTech: The Cornell Hybrid Technology Tool. In P.J. Antsaklis, editor, *Hybrid Systems II*, volume 999 of *LNCS*, pages 265–294. Springer-Verlag, 1995.

[12] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The Next Generation. In *RTSS'95*, pages 56–65, 1995.

[13] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real–Time Systems. *Information and Computation*, 111(2):193–244, 1994.

[14] T.A. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In J.-R. Abrial, E. Borger, and H. Langmaack, editors, *Formal methods for industrial applications: specifying and programming the steam boiler control*, volume 1165 of *LNCS*, pages 265–282. Springer-Verlag, 1996.

[15] L.E. Holloway. Time Measures and State Maintainability for a Class of Composed Systems. In *WODES'96*, pages 24–30, Edinburgh, UK, 1996.

[16] T. Hune, K.G. Larsen, and P. Pettersson. Guided synthesis of control programs using Uppaal. In Ten H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22. IEEE Computer Society Press, April 2000.

[17] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers – Programming Languages, IEC 61131-3*, second edition, November 1998. Committee draft.

[18] S. Kowalewski, O. Stursberg, and N. Bauer. An experimental batch plant as a test case for the verification of hybrid systems. Included in this Special Issue, 2001.

[19] Y. Lakhnech. *Specification and Verification of Hybrid and Real-Time Systems*. PhD thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 1996.

[20] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In E.W. Mayr and C. Puech, editors, *STACS 95: 12th Annual Symposium on Theoretical Aspects of Computer Science, Munich, Germany, March 2–4, 1995*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag, 1995.

[21] K.L. McMillan. *The SMV system*. Carnegie-Mellon University, February 1992. Draft version describing SMV revision 2.2.

[22] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[23] A. Olivero and S. Yovine. *KRONOS: A Tool for Verifying Real-Time Systems. User's Guide and Reference Manual*. VERIMAG, Grenoble, France, 1993.

[24] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming: 5th colloquium, Turin, April 6–8, 1982*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[25] R.S. Sreenivas and B.H. Krogh. On condition/event systems with discrete state realizations. In *Discrete Event Dynamic Systems: Theory and Applications 1*, pages 209–236. Kluwer Academic Publishers, Boston, USA, 1991.

[26] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense time systems. In *Formal Methods 1999 (FM '99)*, volume 1706 (I) of *LNCS*, pages 233–252, Toulouse, France, September 1999. Springer-Verlag.