Development of a Steamboiler Control using Statemate

Lars Kühne Michael Siegel Bernd Weber

University of Kiel
Chair of software technology
Preusserstr. 1-9, 24105 Kiel, Germany
Email: {lku,mis,bwe}@informatik.uni-kiel.d400.de

Abstract. This paper presents a solution to the Dagstuhl steamboiler problem using Statemate[1], a development environment for complex reactive systems. We give a brief introduction to the principles, languages, and tools of Statemate. Afterwards we describe the development process and the final Statemate model of the control program as well as its verification/validation. The paper concludes with an evaluation of Statemate and a comparison with other formal methods that have been applied to the steam boiler problem.

1 Introduction

While *transformational* systems can be described by a function from input to output values, *reactive* systems do not compute functions but perform a continuous interaction with their environment [HP85]. Examples of reactive systems include most kinds of embedded computer systems, control plants, interactive software, and even VLSI design [HLN+90]. In spite of the highly intricate and error prone development process, the complexity and dissemination of reactive systems is rapidly increasing. Without support by appropriate formal methods this trend is particularly alarming because most of these systems have to be classified as safety critical.

The control program of the steam boiler [Abr96] is a typical example of a safety critical, reactive system. The sensors (water-sensor, steam-sensor, pump-controller), the actuators (pumps, valve), and the operator desk constitute the operational environment of the control program. This environment provides information for the control program which as a counter move issues pump and valve commands to keep the water level within safe limits.

Statemate is a collection of tools for the specification, analysis, design, and documentation of complex reactive systems. The worldwide 2000 sold licenses are used with almost equal parity in the fields of aerospace, communications, electronics, and transportation. In this paper we describe our development of the steam boiler control using Statemate and commonly used software engineering techniques [LA90, RBP+91, You89].

[1] registered trademark of i-Logix, Inc.

The development started with a specification of the operational environment of the control system and its verification, i.e. that the modeled units actually behave as required in the original specification. (We use the usually adopted convention that the term 'verification' means 'giving a complete proof' while 'validation' stands for 'giving evidence'). The Statemate model of the environment served as the reference point for the development and evaluation of the steam boiler control. The final *architectural* design of the control system provides complete failure detection and handling. It has been stepwise derived from a considerably simpler *functional* design, which maintains the water level in absence of unit failures.

Besides describing the Statemate models and their evaluation in some detail we also discuss the suitability of Statemate for the development of such systems, based on our personal experience. Theoretical aspects concerning the methodology and the principles underlying Statemate will be discussed as well as pragmatical aspects, such as user interfaces and quality of the implementation. A comparison of various solutions presented at Dagstuhl will be given, comprising formal rigor, verification/validation of the models, and comprehensibility of the formalisms.

The paper is organized as follows. Section 2 gives a brief introduction to the principles, languages, and tools of Statemate. In Section 3 some loose ends of the original specification are clarified. Sections 4, 5, and 6 present the development process and the final Statemate model of the control program as well as its evaluation. Section 7 concludes with an evaluation of Statemate and a comparison with other steam boiler solutions.

2 A Brief Introduction to Statemate

This section summarizes the articles [HLN+90, HdR91] and introduces the concepts needed for the understanding of the following sections. For details we refer the reader to [Har87, HPSS87, ii87a, ii89, ii87b].

Statemate is a collection of tools for the specification, analysis, design, and documentation of reactive real-time systems. Its main principles are to support *executable* specifications, described in a *visual*, yet precise, formalism [HLN+90]. Statemate allows to develop, analyze, and debug diagrammatic descriptions for the *structural*, *functional*, and *behavioral* aspects of the system under development (SUD for short).

The three graphical languages of Statemate correspond to these three aspects. The structural aspect of the SUD, comprising its physical components (modules in Statemate terminology), and their physical links, are described by module charts. As we do not describe the physical realization of the steam boiler control we will not use module charts in this paper, see [ii87a] for details. The functional aspect of the SUD, i.e. its constituting functional entities (activities in Statemate terminology) and their interfaces, consisting of flow of data and control signals, are described by activity charts. Each activity contains a possibly empty hierarchy of sub-activities and at most one designated control activity. It steers the activation and termination patterns of the activities in this hierarchy and is described by a state chart. A state chart can also generate and sense events, sense the activation status of activities, and change values of data items, such as numerical and boolean variables. This when and how the SUD reacts in time constitutes the behavioral aspect of the SUD. The only textual descriptions are used for transition labels and in the data dictionary, where actions, data items, and events are defined.

In this paper we only use activity charts, state charts, and the data dictionary to describe the so called conceptual model [HLN+90] of the steam boiler control. While the functional decomposition of the steam boiler and the respective interfaces between the resulting activities are described in a self-explanatory way by means of activity charts, the state charts formalism and its semantics needs some explanation.

The state charts formalism is an extension of conventional finite state machines. Ordinary state transition diagrams are flat, unstructured, and inherently sequential, causing state explosion when modeling systems with parallel threats of control. In state charts each state consists of a possibly empty hierarchy of state charts modeling possibly concurrent, communicating state transition diagrams (see Fig. 1, 2).

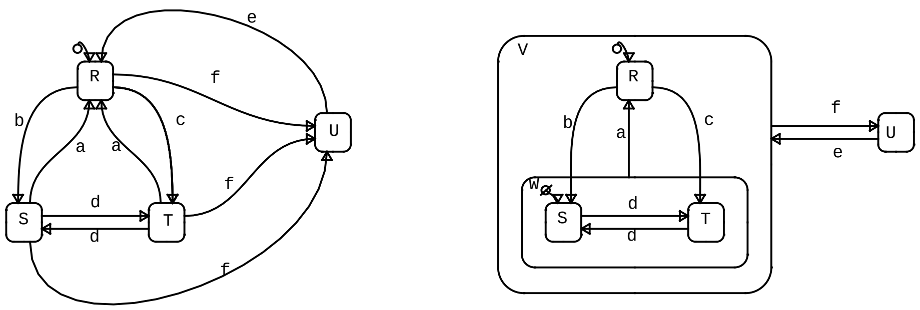

Fig. 1. flat vs. structured description of a state transition diagram. States S and T have been clustered to state W; being in state W means being in state S *or* T. When entering a structured state (see transition E) the default transition indicates which state to enter (in case of transition E this is state R). Transition F corresponds to a high level interrupt: irrespectively of the current state that V is in, the occurrence of F causes that V is left and U entered.

As in the formalism of Mealy machines, output (events in Statemate terminology) can be associated to a transition. Events are the means of communication between parts of a system. Communication is provided by an instantaneous *broadcast* mechanism. Transition labels in state charts have the structure $E[C]/A$ where E is a boolean combination of events, C is a boolean condition, and A is an action. All three parameters are optional. If E occurs and C is true then

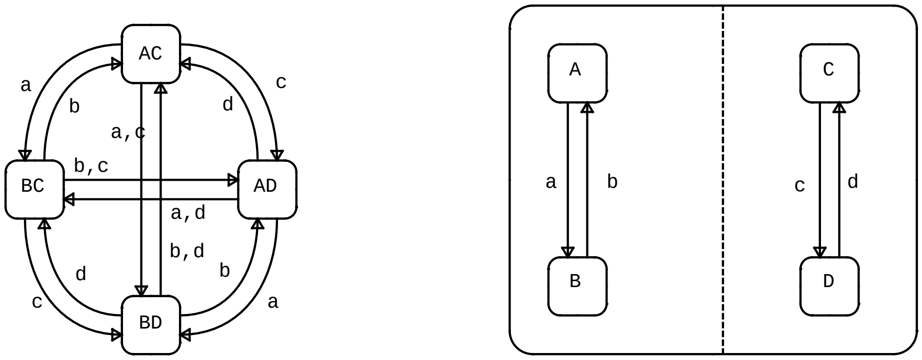

Fig. 2. flat vs. structured description of a state transition diagram. The right figure displays the structure of the system as a parallel composition of two state transition diagrams.

the transition can be taken, resulting in the execution of A. Events, conditions, and actions may refer to the current state of the system. The action part of a transition can generate new events, manipulate variables, and start and stop activities. Timeout events and scheduling actions are available for the specification of timing aspects. For a complete account see [ii87a].

The operational semantics of state charts is a maximal parallelism semantics based on the synchrony hypothesis [BG88]. This semantics is implicitly defined by that part of the simulation tool that performs the stepwise execution of the conceptual model. An informal, though rather precise, description of this algorithm can be found in [ii87b]. It is interesting to note that none of the so far published formal semantics for state charts, see [vdB94] for a comprehensive overview, actually corresponds to the implemented semantics. The heart of the algorithm is to compute the next possible step or in case of non-determinism the set of next possible steps of the SUD. A step is the transition from one legal status of the SUD to another legal status where a status captures the systems currently active states and activities, current values of variables, the set of events generated in the previous step, and history information. To be more precise, the reaction of a system to the generation of a set of events is a sequence of *micro-steps* [HPSS87]. The first micro-step consists of the execution of a maximal set of non-conflicting[2] transitions that are triggered by the events provided by the environment and whose guards evaluate to true in the current status. Subsequent micro-steps consist of maximal sets of non-conflicting transitions that are triggered by the previous micro-step (for readers familiar with state charts semantics: note the singular). A sequence of micro-steps leading to a stable status where no more transitions are triggered constitute a *macro-step* [HPSS87] and correspond to the passage of one time unit on the discrete time scale that un-

[2] two transitions are e.g. in conflict if they depart from the same source state

derlies the execution of the conceptual model. This is a rough description of the *asynchronous* semantics for state charts that we used for the modeling of the steam boiler control; in the *synchronous* semantics, which is not based on the synchrony hypothesis, one micro-step corresponds to the passage of one time unit, see [ii87b] for details.

The described macro-steps can be combined into an operational semantics, following the treatment in [HPSS87].

Further tools of Statemate serve for documentation, code generation (ADA, C, VHDL) and analysis purposes. The available analysis tools are a simulator, used for partial state exploration and thus validation of a system, and a model checker which performs a complete state exploration. Both tools are briefly described in the corresponding Section 6 on the evaluation of the steam boiler model.

3 The Requirements Specification

As also noted by other authors (e.g. [CW96, LL96]), the original specification has some loose ends. In this section we give a list of modifications to overcome these deficiencies. For comparison purposes with other solutions we add a list of failure assumptions that our models rely on in order to perform the control task correctly.

3.1 Modifications to the Original Specification

Concerning the interpretation of the original specification there are two main positions that have been adopted in the submitted solutions: A literal interpretation of the original specification with a corresponding accurate translation into a specific formalism (e.g. [BHW96, BBD$^+$96]) versus a loose interpretation focusing on the general control problem and its solution (e.g. [CW96, LL96, VH96]). We are interested in the development of a steam boiler control which is based on *realistic* assumptions about its operational environment. So, we adopt the second interpretation and thus propose the following modifications:

- On the basis of the available environment information (pump state, pump controller state, water level, current steam output) the general control problem becomes intractable when adopting the *realistic* failure assumption, that faulty environment units deliver *arbitrary* values[3].
  The amount of steam that evaporated during the last cycle is the minimal additional information necessary to detect faults of the water sensor or steam sensor where the delivered values are within the respective static and dynamic limits. Consequently, to cope with the realistic failure assumption we assume that the steam sensor delivers two values: the current steam output and additionally v_{integ}, recording the amount of evaporated steam during the

[3] The original specification relies on the unrealistic failure assumption that faulty components always fail outside their static or dynamic limits.

last cycle. There are two things to note: 1) We do not make any assumptions about the delivered values of v and v_{integ} if only the steam sensor is defective, in particular these values may be within their static and dynamic limits. 2) Omitting v_{integ}, our final model detects and handles all unit failures described in the original specification.

- In order to maximize the available repair time for a component the emergency stop mode has to be entered as late as possible (compare e.g. [LL96]). Thus we strengthen the original specification by introducing an optimal water level which the control program tries to maintain. This optimal level depends on the pump and heater capacity as well as failure probabilities of the physical components.
- There is no need to enter emergency stop mode immediately when the water sensor and another component fail at the same time, since (possibly inaccurate) information about the water level is still available. The program can calculate lower and upper bounds of the water level using the maximal dynamics of the system until a safe level cannot be guaranteed any longer.

3.2 Simplification of the specification

As we aim at a machine supported verification/validation of our final Statemate model, we introduce some further modifications to the original specification in order to reduce the state space of the final model.

- Only one pump/pump controller pair is modeled instead of four. Note that the failure detection for a pump/pump controller pair is independent from the number of pumps in the system.
- As the problem of reliable communication is independent from the original control problem, we do not model the transmission system and assume reliable communication. As a consequence the repair-acknowledgment messages become superfluous.
- There is no reason why the environment, apart from the operator, needs to be informed about the programs current mode of operation and about the detection of unit failures. Thus the MODE and the FAILURE_DETECTION messages are omitted. The information for the operator is modeled by a graphical user interface between control program and operator.

3.3 List of Assumptions

The following list of assumptions is the starting point of our Statemate model. They are a prerequisite for the control program in order to perform its task correctly. This list is meant as a basis for the comparison of our work with other approaches presented in this volume.

- The valve never fails.
- pump and pump controller never fail simultaneously. All other combinations of component failures are allowed.

– In the first cycle after the start of the program all units deliver correct values.
– In the first cycle after a unit has been repaired it delivers correct values.

Note that we do not assume that faulty components only deliver values outside some static or dynamic limits. Only one restriction will be introduced in Section 5 concerning the values of v, v_{integ} and q in case that one component already is defective.

Note further that we do not assume that the actual failure status of the physical components is reported to the control program. This assumption is adopted in some of the papers published in this volume and considerably facilitates failure handling.

4 The Development Process

In this section we describe our development of a specification for the steam boiler control using the Statemate tool and a software engineering approach called 'incremental prototyping' [RBP+91, You89, LA90].

As a first step we successively decomposed the system and its environment. By means of model checking and simulation we have verified that our Statemate model of the environment is consistent with the actual operational environment of the steam boiler control as described in [Abr96]. The final specification of the control program has been obtained by superposition refinement from models with less failure handling capabilities.

The final design and the intermediate models have been verified/validated relative to the environment specification.

4.1 Decomposition of the System

The main goal of the decomposition was to divide the system into small independent units that communicate with their environment exclusively over clearly defined interfaces. The advantage of such a decomposition is the possibility to specify and test the obtained units in isolation. The steam boiler system decomposes – as already indicated in the original specification – into the control program and its environment consisting of the water and the steam sensor, the pump and pump controller, the valve and the operator desk.

After decomposing the system the entire environment has been specified in Statemate. This allowed a scenario analysis of the control problem in the beginning of development phase. On the other hand it facilitates the evaluation of the final model, which can be simulated and tested in a trustworthy model of the real operational environment.

To allow changes of the steam output during simulation an additional HEATER component has been introduced. It can be programmed to change the steam output either user defined or randomly, but it guarantees that the changes only occur within the static and dynamic limits given in the original specification.

The second object added for validation purposes is the boiler containing the water. This object maintains the real water level as a local state variable. It accepts input from all units affecting the water level (pump, valve and steam output) and changes it accordingly. Needless to say that the control program does not have any access to the real water level.

The activity chart in Figure 3 describes the structural view of the steam boiler system, i.e. its decomposition into functional entities and their interfaces.

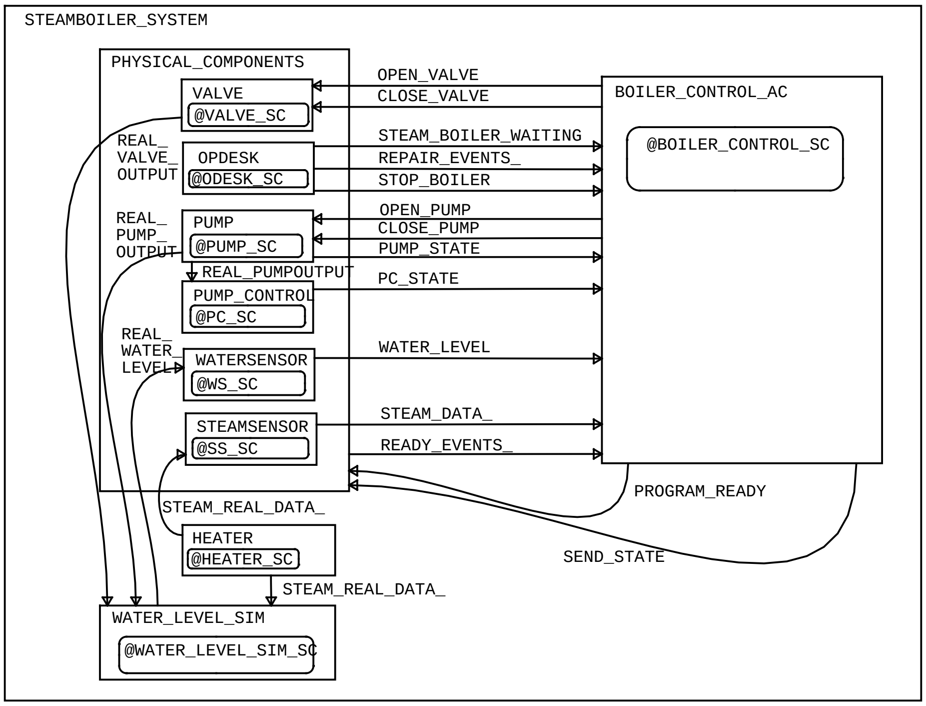

Fig. 3. The Steam boiler Activity Chart

4.2 The Environment

After identifying the various interacting objects in the decomposition phase, the environment objects have been specified.

To validate that the modeled environment behaves like the real operational environment of the boiler control, each physical unit specification has been simulated and tested against the original specification. These module tests (i.e. testing the modeled components in isolation) have been completed by integration tests, which check the interaction and communication between the components. So the second phase of the development process resulted in a faithful model of the operational environment of the control program.

As already mentioned, the control program specification has been derived from simplified variants with less failure handling capabilities. We describe the control program specifications in chronological order of their development. The evaluation of the models is described in Section 6.

4.3 Control Program without Failure Handling

The initial step in the development of the control program was to abstract from all unit failures and to concentrate on the mechanism of steering the boiler in the error-free case. This resulted in a specification of the control program that is able to keep the amount of water in the boiler always within the permitted limits if it can rely on all physical units.

Starting with the error-free case is advisable because we get a clear separation of the different mechanisms to (1) steer the steam boiler collecting environment information and issuing respective pump commands, and (2) to detect and handle failures.

The validated solution for the error-free case has been enriched in the first refinement step by failure detection capabilities.

4.4 Control Program with Simple Failure Detection

In this step we constructed a refined specification of the control program by superimposing a simple failure detection/handling mechanism on the specification for the error-free case. The resulting specification detects all unit failures, where values violating static or dynamic system limits are delivered[4].

It had to be verified that the improved program meets the requirements of the original specification concerning the anticipated faults.

The obtained solution is capable of handling *all* combinations of unit failures in case it gets informed of the failure. The only thing that remains for the next refinement steps is to improve the failure *detection* mechanism of the control program in order to meet our stricter requirements specification.

4.5 Control Program with Maximal Fault Tolerance

In the last step we completed the failure detection of the previous program. Due to the introduction of the integrator for the steam sensor (c.f. Section 3) the final control program is able to detect the delivery of wrong values that are *within* the static and dynamic limits of the respective unit.

The final specification is no longer tractable by the Statemate model checker. The validation effort for this final specification is described in Section 6.

Having described our general approach how to solve the steam boiler problem we next explain the final specification with complete failure detection and handling in some detail.

---

[4] Note that this corresponds to complete failure detection/handling according to the original specification, but not according to our specification

5 The Control Program

Due to space limitations it is impossible to present the complete specification of the control system. Instead we give the top-level view and some representative details.

The control program (Fig. 4) consists of three functional entities that work in parallel. The @-prefix in the state names indicates that the respective state is a super-state. Its structure is described by a state chart with the same name. This mechanism allows high-level design decisions without distracting with the components internal details.

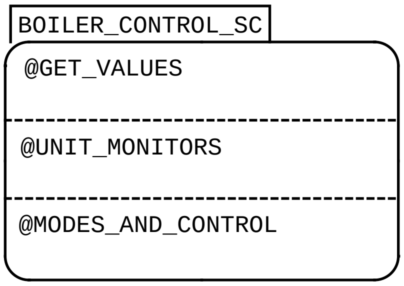

Fig. 4. The BOILER_CONTROL_SC chart

The tasks of these entities are the following:

- GET_VALUES is responsible for periodically reading the current measured values of all physical units.
- UNIT_MONITORS decides which components have to be considered as defective, depending on the received values.
- MODES_AND_CONTROL is responsible for the current operating mode (initialization, normal, degraded, rescue, emergency) of the system. Additionally, it performs consistency checks on the delivered sensor values and issues commands to the pump and the valve.

The state charts description of these entities is given below. Prior to this, the controlling chart for the pump controller activity is presented to illustrate how communication between the control program and its environment works.

5.1 The Pump Controller Chart

The PC_SC chart (Fig. 5) contains three parallel components:

- The ERROR_MODE component determines the current error mode of the pump controller. Events such as PC_NO_VAL and PC_WRONG_VAL are also introduced for every other physical component. They serve as an interface to invoke different component failures during the validation phase.

– The RESPOND_TO_READY component waits for the control program to emit the PROGRAM_READY message and indicates immediately that the pump controller is ready.
– Every time the control program requests the current state of the physical units, the COMMUNICATE_WITH_PROGRAM component responds instantaneously by setting the shared variable PC_STATE according to the current behavior of the pump, taking the current error mode into account. To determine this mode, the SEND_PC_STATE_[5] action (see below) anticipates the effects of occurring error and repair events. If no such event is occurring the program senses the currently active state of the ERROR_MODE component.

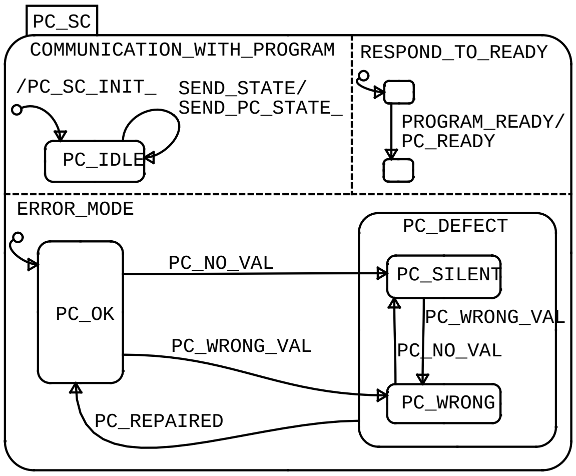

Fig. 5. The PC_SC chart

```
Definition of SEND_PC_STATE_ :
-- First check for the repair event. When it occurs, the reported
-- pc_state is set to the physical pump throughput divided by
-- the pumps capacity (i.e. 1 if the pump is pumping, 0 otherwise)
when PC_REPAIRED then PC_STATE:=P_REAL/PCAP
-- analogously check for other events (NOVALUE is a constant, set to -1)
else when PC_NO_VAL then PC_STATE:=NOVALUE
     else when PC_WRONG_VAL then PC_STATE:=1-P_REAL/PCAP
```

[5] The convention that identifiers ending with an underscore have an underlying textual definition is kept throughout this text as proposed in [Tra90]

```
        else -- no relevant event is occurring, use the active state
             -- to determine the new value of PC_STATE
             if in(PC_OK) then PC_STATE:=P_REAL/PCAP
             else if in(PC_SILENT) then PC_STATE:=NOVALUE
                  else PC_STATE:=1-P_REAL/PCAP
                  end if
             end if;
        end when
    end when
end when
```

The SEND_PC_STATE_ action is performed in one micro-step, consuming no time. Note that every branch of the decision tree performs a write access to the PC_STATE variable. This property will be used in the part of the control program that is responsible for communication with the environment.

5.2 The Get Values Chart

Once the physical units are ready to send values (PHYSICAL_UNITS_READY has been issued), this chart (Fig. 6) is responsible for periodically reading the current sensor values. After every cycle the physical components are requested to send their current state (i.e. the measured values) to the control program. The units answer independently; a constant cycle time and time-consistency of the received values is guaranteed by the fact that all units deliver their values in the same macro-step where the request was issued. This is insured by the specification of the units. As soon as all sensor values have been received[6] this is signaled to the UNIT_MONITORS and the MODES_AND_CONTROL charts.

5.3 The Unit Monitors Chart

The unit monitors (Fig. 7) are the part of the control program that provide the book-keeping about the assumed failure statuses of the physical units.

The unit monitor maintains a separate monitor for every physical unit. When such a monitor is in its OK-state and the GET_VALUES chart signals that the sensors have been read, the monitor evaluates the received values. Whenever this consistency check fails the monitor leaves the OK-state and thereby marks the respective unit as defective.

When the pump works correctly it behaves exactly according to the pump commands given by the program. Thus the program can use the given commands to calculate the values that are expected from pump and pump controller in the next cycle. This is done by the PC_MONITOR and the PUMP_MONITOR. If the pump state or the pump controller state do not match the programs expectations, the corresponding unit is considered defective. In situations where

[6] Statemate offers a simple mechanism to detect a units write access to a data item (e.g. the PUMP_STATE)

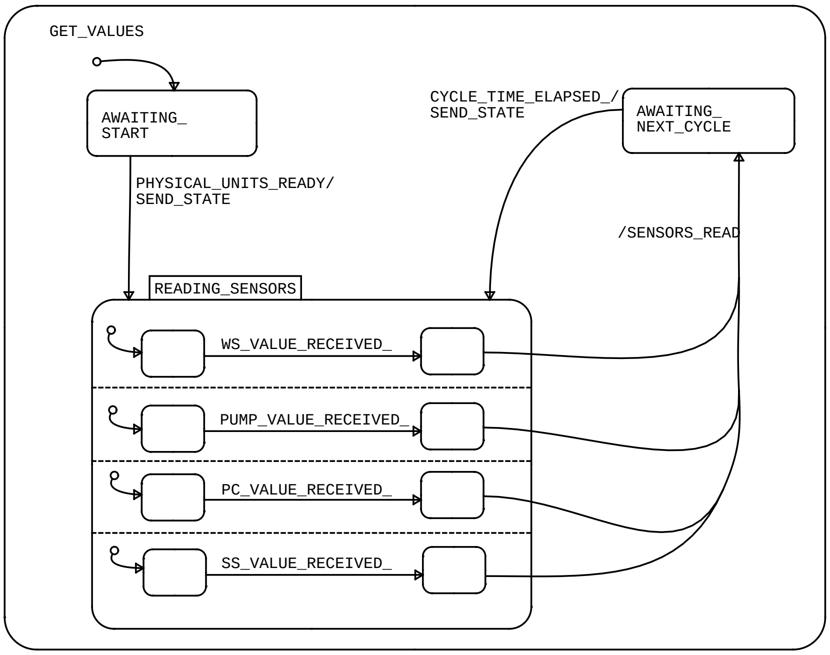

Fig. 6. The GET_VALUES chart; the transition labeled with /SENSORS_READ is activated if and only if all parallel components are in the respective source state of the transition.

it cannot be determined if the pump or its controller is defective, the unit monitor marks both as defective.

In contrast to pump and pump controller, the steam sensor and the water sensor deliver non-discrete values. This makes failure detection more intricate. Failures of steam sensor and water sensor can be separated into three types:

1. violation of static limitations
2. violations of dynamic limitations
3. failures where the delivered values neither violate static nor dynamic system limitations but do not 'fit together'

The unit monitor detects the failure types 1 and 2 by performing the consistency checks listed below. By convention, primed variables refer to the values delivered in the last cycle.

– static limitations:
  - $q \in [0, C]$
  - $v \in [0, W]$
  - $v_{integ} \in [0, W * cycletime]$

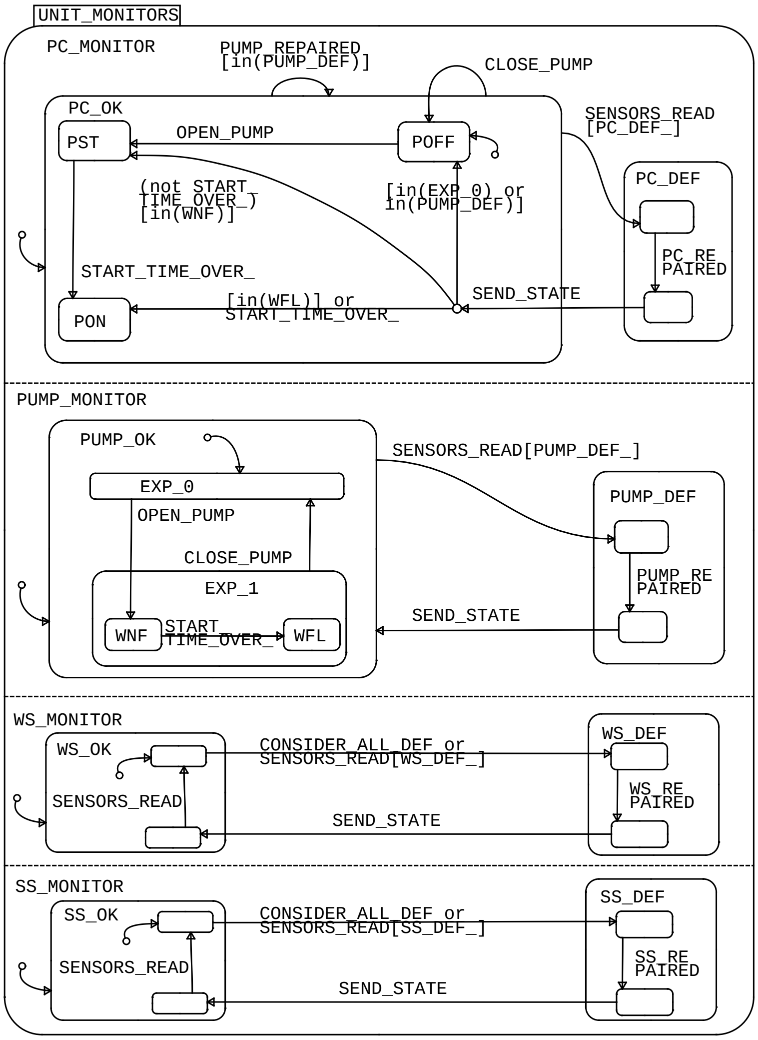

Fig. 7. The UNIT_MONITORS chart. The junction connector (the small circle in the PC_MONITOR) serves as an abbreviation for 3 transitions witch are labeled with SEND_STATE and the respective transition label of the outgoing edge.

- dynamic limitations:
  - $q \in [qc1, qc2]$
  - $v \in [v' - cycletime * U_2, v' + cycletime * U_1]$
  - $v_{integ} \in [vintegc1, vintegc2]$

The last equation reflects that the range of possible values for v_{integ} is limited by the old and new value of v together with v's minimal and maximal gradient $U1$ and $U2$, see Fig. 8.

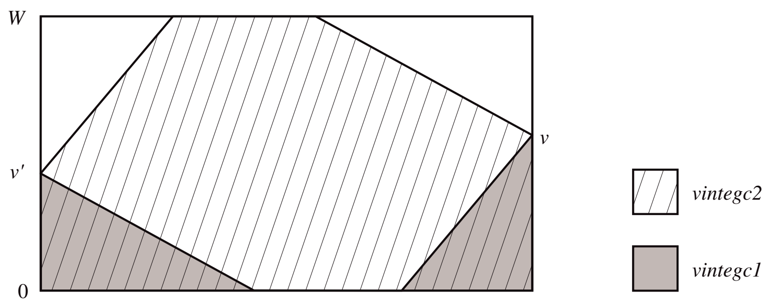

Fig. 8. Limitations for v_{integ}

The negations of the respective conditions yield the definition of 'water sensor defective' (WS_DEF_) and 'steam sensor defective' (SS_DEF_).

One additional check is performed in the MODES_AND_CONTROL component, explained below, in order to detect the third class of failures. If this check fails, a CONSIDER_ALL_DEF event is sent to the unit monitor and pump, water sensor, and steam sensor have to be considered as defective.

5.4 The Modes and Control Chart

In the MODES_AND_CONTROL chart (Fig. 9) a supermode run is introduced that subsumes normal, degraded and rescue mode. After the system has been initialized it enters this run mode. According to the original specification the control program is less fault tolerant in initialization mode than in run mode, so the conditions which lead to an emergency stop are different. Additionally the system can always be forced into emergency stop mode by the operator.

The task performed in INIT mode is simpler than the one performed in RUN mode, so we only describe the latter.

5.5 The Run Chart

As already mentioned, the RUN chart performs additional consistency checks on the values delivered by the physical components. In case every component delivers correct values, the equation

$$q = q' - v_{integ} + \int p \qquad (1)$$

where $\int p$ is the amount of water pumped into the boiler, holds. This allows to detect errors where the steam sensor or the water sensor deliver faulty values within their static and dynamic limits.

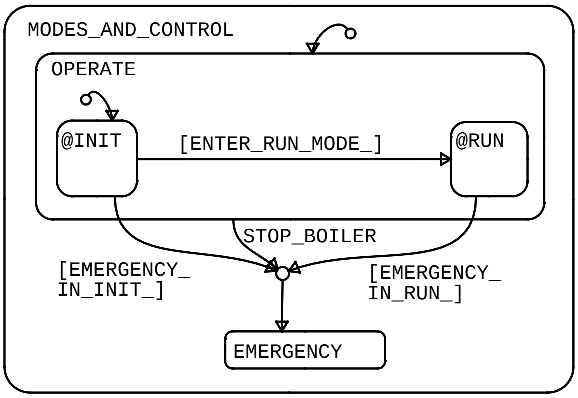

Fig. 9. The MODES_AND_CONTROL chart

It is clear that equation (1) can no longer be used when e.g. the steam sensor is already known to be defective. Splitting equation (1) into the conjunction of the two inequations

$$q \leq q' - v_{integ} + \int p \tag{2}$$

and

$$q \geq q' - v_{integ} + \int p \tag{3}$$

allows further consistency checks, even in presence of faulty components. Their values are then replaced by the minimal or maximal behaviour the unit might have performed.

To guarantee that all failures of the water sensor and steam sensor are recognized another failure assumption is needed, apart from those already presented in Section 3.3: If the water sensor, the steam sensor or the pump is already defective and a further component fails, the delivered values must not satisfy equation (2) or (3). Note that this assumption is not needed to detect the first unit failure.

In case (2) or (3) are violated, obviously the controller program cannot decide which unit has failed. Consequently, the pump, the steam sensor, and the water sensor are considered defective by the program.

The RUN chart consists of two parallel components (see Fig. 10). One is responsible for checking the equations (2) and (3) and issuing pump commands. The other component permanently records the current submode of RUN.

The MODE component is self-explanatory, apart from the fact that the conditions to determine the new mode depend on the current state of the unit monitor.

The CONTROL component consists of a loop which is performed at the beginning of every cycle. After the UNIT_MONITORS have made their decisions the values of defective components are replaced with their maximal behaviour

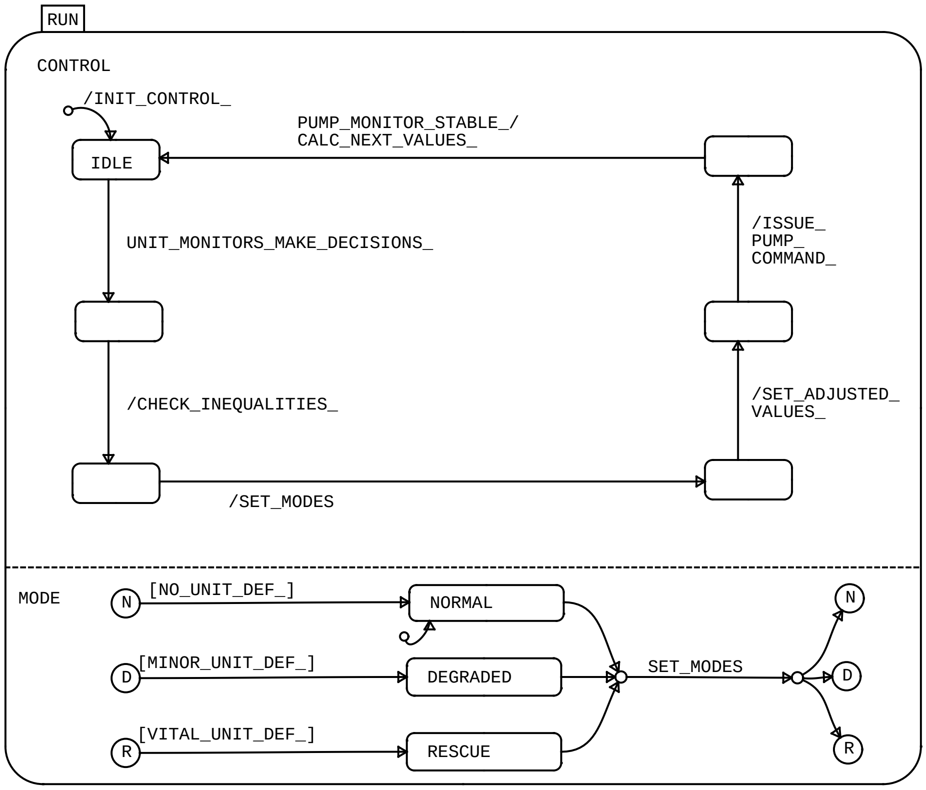

Fig. 10. The RUN Chart; the N,D, and R-circles in MODE are connectors between the right hand side exits and the left hand side entrances to the three substates.

and equations (2) and (3) are checked. If they are violated, the CHECK_IN-EQUALITIES_ action sends a CONSIDER_ALL_DEF to the UNIT_MONITORS which as a result marks the pump, the steam sensor, and the water sensor as defective.

```
Definition of CHECK_INEQUALITIES_:
-- first determine the steam sensor values to use
if in(SS_OK) then
  -- The unit monitor has not discovered a failure of
  -- the steam sensor yet, so the measured values are used
  TEMP_VINTEGC1:=VINTEG;
  TEMP_VINTEGC2:=VINTEG
else
  -- a steam sensor failure has already been detected, so the
  -- minimal and maximal possible values are used
  -- VINTEGC1 and VINTEGC2 have been set by CALC_NEXT_VALUES in the
```

```
  -- previous cycle
  TEMP_VINTEGC1:=VINTEGC1;
  TEMP_VINTEGC2:=VINTEGC2
end if;
-- now determine the pump values
if in(PUMP_OK) and PC/=NOVALUE then
  TEMP_PC1:=PC;
  TEMP_PC2:=PC
else
  TEMP_PC1:=0;
  TEMP_PC2:=PCAP*CYCLE
end if;
-- when the water sensor is defective, the left hand side of the
-- inequations are random, thus we only test when the water sensor seemed
-- OK until now
if in(WS_OK) and
   not (Q-EPSILON<=QA2-TEMP_VINTEGC1+TEMP_PC2 and
        Q+EPSILON>=QA1-TEMP_VINTEGC2+TEMP_PC1    )
then
  CONSIDER_ALL_DEF
end if;
```

After reacting on the generation of the CONSIDER_ALL_DEF event the UNIT_MONITORS become stable for the rest of the cycle and the new operating mode can be determined. In the RUN chart the adjusted values ($qa1$, $qa2$ etc.) are set to determine whether the pump commands have to be sent.

```
Definition of ISSUE_PUMP_COMMAND_:
if not (in(PUMP_DEF)) then
  -- commands only necessary if pump not defective
  if (QA1+QA2)/2<QOPT then
    -- the water level is below its optimal value
    if in(EXP_0) then  -- the pump is not turned on (see UNIT_MONITORS)
      OPEN_PUMP;
      -- T is the time since the pump has been opened. It is
      -- used in CALC_NEXT_VALUES_ to calculate the pumps throughput
      -- in the next cycle when the pump is not pumping
      T:=0
    end if
  else
    -- the water level is above or equal to its optimal value
    if in(EXP_1) then
      -- the pump is turned on (see UNIT_MONITOR)
      CLOSE_PUMP
    end if
  end if
end if
```

Finally the maximal behaviour of every component is calculated to allow failure detection after the next cycle.

```
Definition of CALC_NEXT_VALUES_:
VC1:=MAX(VA1-U2*CYCLE,0);
VC2:=MIN(VA2+U1*CYCLE,W);
-- A is the time needed to reach maximal steam output
-- B is the time needed to reach no steam output
A:=(W-VA2)/U1;
B:=VA1/U2;
VINTEGC1:=(0.5*B*VA1+(CYCLE-B)*0.5*VC1);
VINTEGC2:=(0.5*A*(W+VA2)+0.5*(CYCLE-A)*(W+VC2));
-- calculated minimal Q at next cycle
QC1:=QA1-TEMP_VINTEGC2;
-- calculated maximal Q at next cycle
QC2:=QA2-TEMP_VINTEGC1+CYCLE*PCAP;
-- calculate throughput of the pumps within next cycle
if in(EXP_0) then
  PC:=0;
end if;
if in(WNF) then
  PC:=MAX(0,T+CYCLE-PUMP_START_TIME)*PCAP;
  T:=T+CYCLE   -- set T for next cycle
end if;
if in(WFL) then
  PC:=CYCLE*PCAP
end if;
if in(PUMP_DEF) then
  PC:=NOVALUE
end if
```

Note that, according to the state charts semantics, all actions are performed instantaneously without passage of time when the respective transition is taken.

In this section we have presented some details of the controller specification. For a complete specification please contact one of the authors. In the next section we summarize our verification/validation of the Statematemodels for the controller and its environment.

6 Verification and Validation of the Model

One main advantage of using Statemate as a specification tool is the fact that the derived specifications are executable. Exploiting this, Statemate offers two analysis tools to check safety properties that are expressible as a reachability problem:

- the dynamic test tool (= model checker) performs a complete state exploration. It is consequently used for the verification of safety properties.
- the simulation tool performs a partial state exploration and is thus used for the validation of safety properties.

The heart of both tools is the ability to carry out steps of the SUD's dynamic behaviour. Following the traditional approach of software evaluation we used dynamic tests to verify the functional entities depicted in Fig. 3 in isolation. As the final model of the system was unmanageably complex for the dynamic test tool, an integration test was done using the simulation tool.

6.1 Dynamic Testing

Given a conceptual model (i.e. a number of state charts and activity charts) and a number of state predicates, the dynamic test tool searches the entire state space for a configuration which satisfies at least one of the predicates. Thus, to prove invariance of a state assertion for a specification, its negation is fed into the dynamic test tool which then reports whether it is reachable or not.

For the test of the water sensor and steam sensor we replaced the dense domain by a discrete approximation. Table 1 lists the properties that have been verified for the respective components as well as technical data about the proofs.

Since the control program is closely interacting with its environment, it was impossible to make a stand-alone verification using the dynamic test tool.

For the integration test we also tried to use the dynamic test tool but due to the size of the state space (approx. $4 * 10^{12}$ states) the tool did not succeed in finishing the test within 3 days of computation on a SPARC 20/712.

6.2 Program Controlled Simulation

Most simulation tools only offer an interactive simulation mode, where the user triggers the performing of steps and simultaneously inspects the resulting behaviour of the system for correctness. Statemate additionally offers a program controlled fully automatic simulation for the evaluation of large systems. In this case a program written in a high level language called SCL (simulation control language) replaces the user. SCL programs employ variables and support the usual control structures; a main loop initiates the performing of steps. The distinctive concept of SCL with respect to validation purposes is the breakpoint construct. Breakpoints cause the specifications execution to stop and take appropriate actions when particular situations come up. We used breakpoints to

- set the actual steam output
- generate unit failures and repair events with a given probability
- output statistics into a file
- survey that all safety properties are satisfied

Two breakpoint examples are given below. The first breakpoint randomly generates pump controller failures while the second breakpoint checks if the real water level is between $M1$ and $M2$, and records violations of this condition in a file.

```
SET BREAKPOINT every 1 DO
  -- invoked after every step
  IF rand_Iuniform(1,80) = 1 and (not in(pc_def)) THEN
    -- if pump controller is not already defective then with
```

unit	verified properties	cpu time
water sensor	the water sensor sends the right value if it is not defective `wr(Q)[in(WS_OK)] implies Q=Q_REAL` if sending of no value is requested then the water sensor actually does not send a value `wr(Q)[in(WS_NO_VAL)] implies Q=NOVALUE`	00:16:13
pump con troller	the pump controller sends the right value if it is not defective `wr(PC_STATE)[in(PC_OK)] implies PC_STATE=P_REAL/PCAP` if sending of no value is requested then the pump controller actually does not send a value `wr(PC_STATE)[in(PC_NO_VAL)] implies PC_STATE=NOVALUE` if the pump controller is defective it sends the wrong value `wr(PC_STATE)[in(PC_DEFECT)] implies P_REAL/PCAP/=PC_STATE`	00:10:03
steam sensor	the steam sensor sends the right value if it is not defective `wr(V)[in(SS_OK)] implies V=V_REAL and VINTEG=VINTEG_REAL` if sending of no value is requested then the steam sensor actually does not send a value `wr(V)[in(SS_NO_VAL)] implies V=NOVALUE and VINTEG=NOVALUE`	01:34:19
pump	the pump sends the right value if it is not defective `not PUMP_DEFECTIVE_ implies PUMP_STATE=POUT_REAL/PCAP` the pump reacts correctly to failure invocation `wr(PUMP_STATE)[in(PUMP_NO_VAL)] implies PUMP_STATE=NOVALUE` `wr(PUMP_STATE)[in(PUMP_WRONG_VAL)] implies PUMP_STATE=1-POUT_REAL/PCAP`	11:14:14
system	in run mode the water level stays between $M1$ and $M2$ `[in(RUN)] implies M1<=q_real and q_real<=M2`	>72:00:00

Table 1. Dynamic Test results. All checks were run on a SPARC 20/712, 320 MB physical RAM. All properties but the last were verified, the system property check was stopped after 3 days.

```
    -- probability 1/80 destroy it by sending an appropriate event
    PC_NO_VAL
  END IF
END BREAKPOINT;

SET BREAKPOINT [ (q_real < M1 or M2 < q_real) and in(RUN) ] DO
  write(logfile,'ERROR safety condition in step ',step_number,'\n');
  -- the step_number is automatically increased by Statemate
END BREAKPOINT;
```

Reachability of states violating the following safety conditions of the system was checked.

- the program is in normal, rescue or degraded mode and $q_{\text{real}} \notin [M1, M2]$
- $q_{\text{real}} \notin [qc1, qc2]$

– a defective component has not been detected after one cycle
– the control program considers a unit defective and all units are OK

Table 2 summarizes the simulation results.

Program length	514 lines
Program run time (SPARC 20/712)	72 cpu-hours
simulated cycles	450000
simulated steps	12.6 mio.
number of pump failures	21963
number of pump controller failures	14179
number of water sensor failures	20134
number of steam sensor failures	11088
number of emergency stops	242
number of reached conditions	none

Table 2. Automatic simulation results

6.3 Interactive Simulation

Besides the program controlled simulation using SCL programs there exists an interactive simulation mode that was used during the development process. With this simulation type, specifications can be executed step by step or run continuously. In both cases initiating steps of the executed specification and monitoring its output is left to the user. It is possible to bind data items, conditions, events and states to graphical objects on a panel to support this monitoring, see Fig. 11. According to our experience interactive simulation provides a convenient debugging mechanism, as it allows scenario analysis during the development phase of a specification.

7 Evaluation, Comparison and Conclusion

7.1 Evaluation of Statemate

The evaluation is split into two parts: (1) the extent to which Statemate covers the life-cycle of reactive systems and (2) a discussion of the main principles of Statemate.

Coverage of the system-life-cycle: In contrast to most formal methods investigated in the research community, Statemate covers most aspects of the system-life-cycle of reactive systems. Besides providing specification languages for system analysis and system design, there are code-generators (ADA, C,

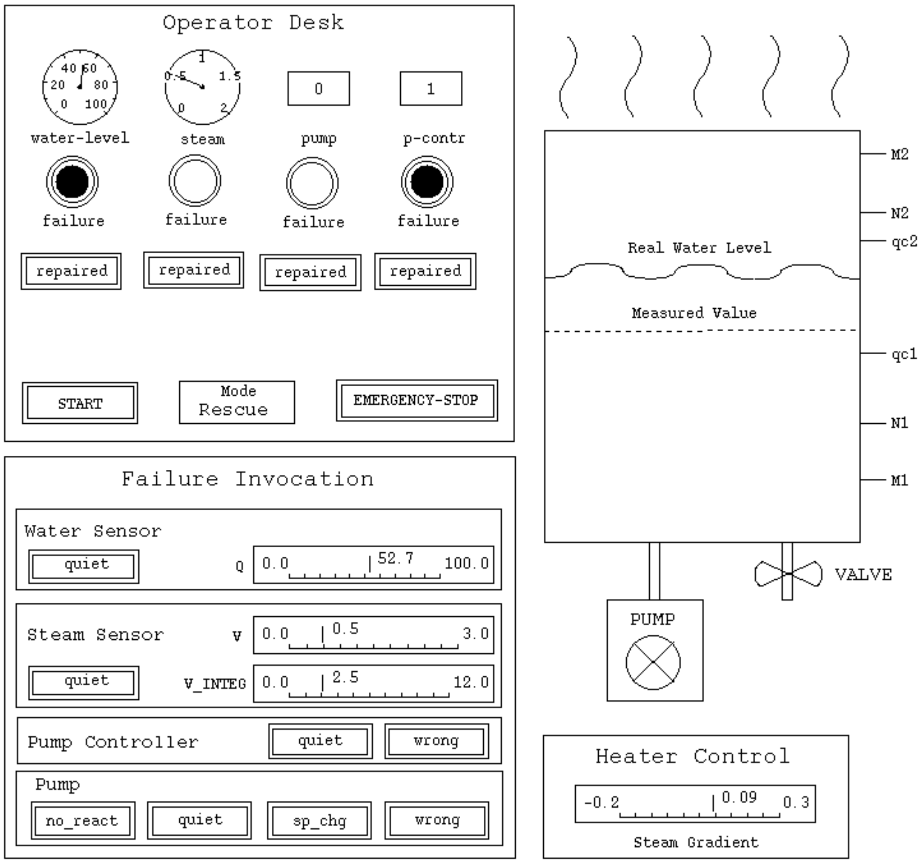

Fig. 11. The steam boiler panel. The upper left corner shows the interface for the operator, in the lower left corner unit failures can be invoked. The right hand side displays a visualization of the current steam boiler status.

VHDL) as well as features for quality assurance and project management (documentor, requirements tracibility tool, report tool, consistency mechanisms, and version control for data bases).

However, two main aspects are missing: As Statemate is a.o. used for the development of controller in mass products (consumer electronics, cars, communication devices etc.), where every bit counts, the automatically generated code is in most cases unsuitable, resulting in hand coding of the Statemate specifications. As this situation will not change in the near future, tools to support module and integration testing are necessary to bridge the gap between specification and code. Secondly, Statemate only supports explicit specification. To cover the requirements specification phase in the software-life-cycle, more abstract behavioural specification formalisms and supporting tools are necessary. An interesting extension in this direction, using symbolic timing diagrams, is presented in [DH94].

Discussion of main principles: The first distinguishing feature of Statemate, compared to most industrially used software development methods (see [Dav90] for an overview), is *executability.* It allows, in case of controller development, for scenario analyses of the control problem, for quick debugging of initial designs by means of interactive simulation, and for rapid prototyping. Furthermore, executability supports algorithmic analysis of specifications using elaborated simulation techniques (s.a. SCL-simulation) and model checking.

The second major principle is the support of graphical specification languages. Statemate allows for natural, concise specifications, because the two important structuring mechanisms for complex, reactive systems, hierarchy and concurrency of tasks, are the basic constructors of all three languages.

The most studied language, state charts, it very powerful when it comes to modeling of control flow. Beside inter-level transitions and history mechanisms it contains convenient real-time constructs (timeout, scheduling). The power of this language involves a problem as already mentioned in Section 2: there does not exist an agreed on formal semantics for state charts. This impedes the development of a tractable compositional refinement notion for Statemate specifications, the development of add-on tools, and the extension of the current framework by e.g. a requirement specification language.

In a nutshell, our personal conclusion is: Despite the explained draw backs of Statemate and its sometimes disturbingly bad implementation (s.a. man-machine interface, storage leak in the simulation tool, very inefficient model checker, unsatisfactory SCL-compiler, just to mention some), Statemate is a *very* valuable tool for the development of complex reactive systems, because its underlying principles are sound and convincing.

7.2 Comparison With Other Approaches

Several introductory remarks are necessary: We present a comparison of *how the steam boiler problem* has been solved in [BBD+96, BHW96, BSS96, Ber96, CW96, LL96, Sch96b, Sch96a, VH96][7]. The different skill and experience of the various teams that submitted these solutions makes it difficult to draw generally valid conclusions about the used methods and formalisms; and a more fundamental comparison on the basis of the available literature is clearly beyond the scope of this section. Several other factors tend to complicate the comparison:

- The (purposely) rather fuzzy task that were to solve. So solutions on several abstraction levels have been elaborated, ranging from behavioral specifications [CW96, Sch96a, VH96] (i.e. collections of desirable properties), over explicit specification [BHW96, BBD+96, CW96, Sch96a, LL96, BSS96, VH96] (e.g. as action systems or explicit state transition diagrams), to implementations in some programming notation [Sch96a].

[7] only these papers were sent to us by October first upon request.

- The various teams put their emphasis on different aspects of the system: e.g. performance properties [LL96], accurate modeling of time [CW96], completely machine checked proofs [VH96], reification [Sch96a], complete failure handling [BHW96, CW96].
- Two different ways of interpreting the original specification: A literal interpretation with a corresponding accurate translation into a specific formalism [BHW96, BBD+96] versus a loose interpretation focusing on the general control problem and its solution [CW96, LL96, VH96].

The main criteria that we used to evaluate and compare the different approaches are derived from the intended role of formal methods for the development of complex reactive systems: to provide a means for *verified/validated*, *comprehensible* specifications.

All of the evaluated approaches can be termed "formal" in the sense of having a formal semantics. The formal semantics is the very basis for any kind of verification/validation effort. Now it is interesting to note, that most submitted solutions confine themselves to the *modeling* of the steam boiler control without subsequent evaluation of their model. In several of these papers we found inconsistencies and bugs which have been communicated to the authors. This shows that without a careful evaluation of the model, formal methods are only a means for structuring a designers thoughts but they are inadequate when it comes to preparing *reliable, consistent* specifications that satisfy the requirements and behave over time as expected.

Verification/Validation: An exemplary evaluation can be found in [VH96]. Starting from a behavioral specification, the control program is derived by stepwise refinement within a mixed formalism that contains specification as well as programming constructs. Each refinement step has been completely machine checked using the theorem prover PVS. The machine supported evaluation of our solution using model checking and SCL-simulation has been presented in the previous section. It is fair to note that the Statemate evaluation tools are almost self-explanatory while system evaluation using PVS is quite intricate. Besides these two machine supported evaluations, hand-made evaluations can be found in e.g. [LL96, BHW96, BSS96]. A derivation of a control program using Back's refinement calculus [Bac90] is sketched in [BSS96]. The abstraction relations between the different specifications are given, but no proofs. An interesting evaluation is sketched in [BHW96], where a state charts description of the system is used to derive proof obligations about the interplay of Z-schemes which are associated with transitions. The obtained proof obligations seem manageable. In [LL96] refinement proofs using possibility mappings are given. Though hand made evaluations/proofs add reliability concerning correctness of the specification they are also error prone and thus lack the formal rigor that is obtained by machine checked proofs. In several of the submitted solutions it is stated that after the system has been modeled proofs could in principle be carried out for system evaluation. We believe that the main question has not been whether it is *possible* to model the system within a specific formalism, but to demonstrate that the applied formalism is able to cope with the complexity of this system, to yield manageable proofs and a finally debugged design. In our case it took approximately 20% of the time to come up with the various specifications and 80% to evaluate and debug the original products. When modeling all details mentioned in the original specification, the complexity of the control system becomes high enough that bugs and inconsistencies in unevaluated specifications are almost unavoidable.

Machine support for proofs has the additional advantage that all proof assumptions (faults, boundaries of variables etc.) have to be stated explicitly (see e.g. [VH96]) because otherwise the proof fails. This reveals on the one hand mandatory extra assumptions for the correct working of the specified system and on the other hand eases the task of comparing different approaches with respect to the assumptions that were made. Unfortunately, most of the Dagstuhl submissions lack a complete list of assumptions.

What has been modeled: A lot of differences can be observed when it comes to modeling the control system. With respect to faults the assumptions range from completely fault-free components [LL96], over single faults in isolation (only transmission failures or only water sensor failure[VH96]) to complete failure handling e.g. [BHW96, CW96]. In the case of complete failure handling it is interesting to note that in some approaches global knowledge about the failure status is assumed in the control program.

Comprehensibility: Besides yielding debugged, consistent specifications, a second not less important aspect of a formal method is comprehensibility of the models. An industrially usable formal method has to serve as a communication medium between customer, designer, and programmer. It is our belief that for the specification of control flow graphical approaches, based on structured automata, yield far more readable specifications than non-graphical methods. The advantage of graphical formalisms does not show up that clearly in the elaboration of the steam boiler control program. This is due to the fact that the heart of the control program consists of one main control loop only. This control loop can easily be coded in non-graphical formalisms. In this respect the steam boiler is not a perfect benchmark to evaluate formal methods for the development of reactive systems. The control flow for monitoring the failure statuses of the units is more complex and results in some combinatorial complexity when coding the control flow.

Due to the mentioned reasons (skill and experiences of the teams, different emphasis, different failure treatments and assumptions) we did not evaluate the formalisms for time consumption.

Conclusion: We conclude with the (subjective) observation that a considerable part of the evaluated submissions did not succeed in convincingly demonstrating the practical applicability of the advocated approach because a thorough evaluation of the models is missing.

None of the used formalisms seems capable yet to provide a solid basis for the development of correct, comprehensible specifications for safety critical applications. Omissions are a.o. tool support for the evaluation of the specifications, separation into structural, functional and behavioural aspects of the system to gain comprehensibility, prototyping facilities, and testing facilities for integration and system tests. In the case of Statemate, which covers all but the last point, we consider the missing link between different specifications, i.e. a compositional refinement notion and respective tool support, to be the main omission.

Acknowledgment: We wish to thank the organizers of the Dagstuhl Case Study. Elaborating non-toy examples is the best basis for a comparison of different formal methods and they reveal weak points of the currently used techniques, thus clarify necessary future work to completely cover the system life cycle by formal methods.

References

[Abr96] Jean-Raymond Abrial. Steam boiler control specification. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996.

[Bac90] R. J. R. Back. Refinement calculus: Parallel and reactive systems. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *LNCS*, 1990.

[BBD+96] Christoph Beierle, Egon Börger, Igor Durdanovic, Uwe Glässer, and Elvinia Roccobene. An evolving algebra solution to the steam boiler control specification problem. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996.

[Ber96] Pascal Bernard. A Z specification of the steam boiler. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[BG88] Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. Technical report, Ecole National Superieure des Mines de Paris, 1988.

[BHW96] Robert Büssow, Maritta Heisel, and Matthias Weber. A steam boiler control specification with statecharts and Z. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[BSS96] Michael Butler, Emil Sekerinski, and Kaisa Sere. Steam boiler specification and refinement. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[CW96] Jorge Cuellar and Isolde Wildgruber. The Dagstuhl steam boiler controller problem: The TLT solution. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[Dav90] Alan Davis. *Software Requirements*. Prentice Hall International Editions, 1990.

[DH94] Werner Damm and J. Helbig. Linking visual formlisms: a compositional proof system for statecharts based on symbolic timing diagrams. In *Working Conference on Programming Concepts, Methods and Calculi, San Miniato, Italy*. IFIP, North-Holland, 1994.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[HdR91] Cornelius Huizing and Willem Paul de Roever. Introduction to design choices in the semantics of statecharts. *Information Processing Letters*, 37, 1991.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), 1990.

[HP85] David Harel and Amir Pnueli. *Logics and Models of Concurrent Systems*, chapter On the development of reactive systems. NATO Advanced Science Institute Series F: Computer and Systems Sciences 13. New York: Springer Verlag, 1985.

[HPSS87] David Harel, Amir Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of Statecharts. Proceedings of the 2nd IEEE Symposium on Logic in Computer Science. IEEE Press, 1987.

[ii87a] i-Logix inc. The languages of statemate. Technical report, i-Logix inc., Burlington, MA., 1987.

[ii87b] i-Logix inc. The semantics of statecharts. Technical report, i-Logix inc., Burlington, MA., 1987.

[ii89] i-Logix inc. The statemate approach to complex systems. Technical report, i-Logix inc., Burlington, MA., 1989.

[LA90] Shem Tov Levi and Ashok Agrawala. *Real Time System Design*. McGraw Hill International Editions, 1990.

[LL96] Gunter Leeb and Nancy Lynch. Using timed automata for the steam boiler controller. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Sch96a] Michael Schenke. The steam boiler: From Duration Calculus to OCCAM. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[Sch96b] Christian Schinagl. Steam boiler control specification problem: An attempt using RSL. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[Tra90] Trachtmann. Elements of statemate style, a framework for embedded system specification. unpublished, 1990.

[vdB94] Michael von der Beeck. A comparison of statecharts variants. In Hans Langmaack, Willem Paul de Roever, and Jan Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 863 in LNCS. Springer, 1994.

[VH96] Jan Vitt and Jozef Hooman. Specification and verification of a real-time and fault-tolerant steam boiler control system using PVS. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Steam Boiler Case Study*, LNCS. Springer, 1996. to appear.

[You89] Edward Yourdon. *Modern Structured Analysis*. Prentice Hall International Editions, 1989.

This article was processed using the LaTeX macro package with LLNCS style