Specifying and Verifying the Steam Boiler Control System with Time Extended LOTOS

Andreas Willig
Technical University Berlin
tel.: (030) 314 23 831
e-mail: Willig@ftsu00.ee.tu-berlin.de

Ina Schieferdecker
GMD Fokus
tel: (030) 254 99 241
e-mail: ina@fokus.gmd.de

January 5, 1996

Abstract

The paper presents a specification of the steam boiler system in Time Extended LOTOS as an example to describe real-time, hybrid systems containing parts with discrete and continuous behavior in a time extended process-algebraic formalism.

The specification has been developed in three design steps — the specification of functional behavior in LOTOS, the specification of timed behavior in TE-LOTOS (an upward-compatible time extension of LOTOS), and the specification of data dependencies in TE-LOTOS with Gofer (a functional data type language).

The resulting specification is very modular, succinct, and easy to read. Most importantly, it is an implementation-near specification. It offers means for logical reasoning on the system properties, for stepwise refinement, simulation and prototyping of the system.

Contents

1 Introduction

The steam boiler control problem has been given as a case study at the 1995 Dagstuhl seminar on methods and semantics for specification [ABL95]. It aimed at a critical assessment of the advantages and drawbacks of formal methods for practical real-time applications. Although it is a small sized specification problem, it covers critical aspects from practical real-time systems.

The steam boiler system that is informally described in [ABL95], is an example for a safety-critical, real-time system. Normally, the steam boiler is filled with water for the generation of steam. There are four pumps to fill water into it. Furthermore, two measuring devices measure the level of water and the quantity of steam leaving the steam boiler through an outlet on its top, respectively. The physical system (`PS`) is controlled by a control unit (`CU`) that is responsible for regulating the level of water in between a lower and an upper bound. If the level of water exceeds one of these bounds for more than 5 sec, the steam boiler explodes. In normal operation, `CU` reads every 5 sec the quantity of water and steam and information on the pump state. If the water level is going too low, the pumps are opened by `CU`; if it is going too high, the pumps are closed. A closed pump needs 5 sec to open, but an open pump closes immediately. `CU` communicates via messages with the physical system over an unreliable transmission medium. If `CU` detects a transmission error (e.g. a requested message does not occur in time or an unexpected message is received) or when the level of water is approaching the bounds, it has to enter the emergency stop mode immediately, which is assumed to be safe. In the case of a defective pump or measuring device, `CU` should enter the rescue or degraded mode, respectively. It estimates the level of water in order to keep it within the bounds. If defective devices are repaired, `CU` can continue in the normal mode.

The control unit `CU` has to work correctly in order to prevent the level of water exceeding the bounds for more than 5 sec. Otherwise, the steam boiler system or the turbine in front of it might explode. Thus, the steam boiler system has to meet two main safety requirements

1. In the case of the absence of severe failures in physical devices and in the transmission medium, the water level in the steam boiler has to remain within the given bounds except of transient periods with a maximal duration of 5 sec.

2. In the case of any severe errors, the steam boiler system has to enter the emergency stop mode in order to prevent disastrous things to happen.

The steam boiler system is an hybrid system that contains parts with continuous behavior (some of the physical devices) and with discrete behavior (the control unit). According to the safety requirements hard deadlines have to be met.

This paper considers the specification of the steam boiler within TE-LOTOS — Time Extended LOTOS [LL94, LLdFQ95]. LOTOS has successfully used for the description of communication protocols. Its formal semantics is the basis for a well-founded design methodology ranging from abstract to concrete specifications via stepwise refinement, offering means to formal reasoning, prototyping, and testing. With the introduction of a time concept into LOTOS, it becomes also applicable to the design of real-time systems. This paper presents one attempt to gather practical experiences for this new application area of LOTOS. The specification of a real-time system comprises of

1. the system architecture, i.e. its structuring into components and subsystems with their interfaces and the relation between them and to the environment,
2. the functional behavior of the components and subsystems, and
3. the timing constraints that have to be met for a correct and safe system execution.

TE-LOTOS offers several feature for the specification of real-time systems:

1. the definition of processes and interaction points (the so-called gates),
2. the parallel or sequential composition of processes including the definition of proper synchronization between them,
3. the definition of behavior expression that represent the externally visible and internal behavior of processes as sequences of actions, and
4. the definition of timing constraints that impose time windows for synchronization or the passage of time.

We used a series of refined specifications in order to introduce step by step the several constraints on the behavior of the steam boiler system. Firstly, we considered only the functional behavior and specified the steam boiler system in LOTOS [ISO88]. This functional specification itself resulted also from a number of refined LOTOS specifications. The functional design step has been formally verified for its correctness by the use of LOTOS tools. A number of tests, which represent safety conditions for the steam boiler system, were applied to the functional specification.

Afterwards, we introduced the timed behavior by the use of the timed operators of TE-LOTOS [LLdFQ95], which is an upward-compatible time extension of LOTOS. Finally, we specified the data dependencies of the steam boiler system in the functional data type language Gofer [Jon93] and integrated them into the time-extended functional specification. Due to the lack of tools for TE-LOTOS, the later two steps have been manually checked for their correctness.

The resulting specification in TE-LOTOS with Gofer gives an architectural design for the steam boiler system. It defines *how* the requirements of the steam boiler system are fulfilled, i.e. the requirements are implicitly reflected in the specification. The specification is implementation-near and can be used for building a system prototype. In addition, the requirements are explicitly defined in the tests. The tests define *what* requirements have to be met. The tests are applied to the specification in the test approach to verify the correctness of the specification.

The paper is structured as follows. The paper mainly adheres to the three design steps. The functional specification and its verification is considered first. The subsection on verification describes the testing approach that was used to check the correctness of the specification. Afterwards,

Name	**Syntax**	**Explanation**
Stop Process	$stop$	Deadlock
Exit Process	$exit(e_1 \dots e_n)$	Termination with value passing
Action prefix	$ge_1 \dots e_n[SP]; Q$	Observable action g with value offering $e_1 \dots e_n$ and communication guard SP followed by behavior Q
Internal action prefix	$i; Q$	Internal action i followed by behavior Q
Choice	$Q_1 [] Q_2$	Choice between behaviors Q_1 and Q_2
Parallel Composition	$Q_1 [g_1 \dots g_n] Q_2$	Parallel execution of behaviors Q_1 and Q_2 with synchronization in gates $g_1 \dots g_n$
Hiding	$hide\, g_1 \dots g_n\, in\, Q$	Make $g_1 \dots g_n$ invisible and unaccessible from outside
Enabling	$Q_1 >> accept\, x_1 : s_1 \dots x_n : s_n\, in\, Q_2$	Sequential composition of Q_1 and Q_2 with value passing to Q_2 after termination of Q_1
Disabling	$Q_1 [> Q_2$	Interrupting Q_1 by Q_2
Process Instantiation	$P[g_1 \dots g_n](e_1, \dots, e_m)$	Executing the behavior Q with actual gates $g_1 \dots g_n$ and actual parameters $e_1, \dots e_m$ of process P
Guarding	$[SP] -> Q$	Executing Q if guard SP evaluates to true
Value declaration	$let\, x_1 : s_1 = e_1 \dots x_n : s_n = e_n\, in\, Q$	Bind the values $e_1 \dots e_n$ to the variables $x_1 \dots x_n$ resp. in behavior Q

Table 1: LOTOS behavior expressions

the time-extended specification is discussed. The data-extended specification is considered at last. Each of this three parts is introduced by a short overview on the used specification features and formalisms. Subsequent to that, we give answers to the questionnaire given [ABL95] in order to evaluate the presented approach. Conclusions identify advantages and drawbacks of TE-LOTOS with Gofer for the specification and verification of real-time systems.

2 The Functional Specification in LOTOS

2.1 Overview on LOTOS

The formal description technique LOTOS — the Language Of Temporal Ordering Specification [ISO88] — has been developed by ISO for the unambiguous definition of the functional behavior of information processing systems. It is based on process-algebraic calculi for the description of behavior, in particular on Milner's CCS — the Calculus of Communicating Systems [Mil80] — and on Hoare's CSP — the calculus of Communicating Sequential Processes [Hoa85]. Data dependencies are described in the algebraic data type language ACT ONE [EM85]. Due to historical reasons, LOTOS has primarily used for the specification of protocols within OSI and for verifying their functional correctness.

A LOTOS specification defines the system behavior as the temporal order of externally visible events at the system interface. The basic notions of LOTOS are *actions* that represent atomic and instantaneous system functionalities, and basic *processes* (stop and exit) that represent deadlock and termination, respectively. The occurrence of an action is called *event*. Actions and processes can be composed by a number of operators such as action prefixing or parallel composition in order to build complex behavior expressions. For an overview on LOTOS behavior expressions please refer to Table 1. More complex processes can be defined by behavior expressions. The interface of a process is defined by gates that identify externally visible and accessible actions. Processes can built up a process hierarchy. The *recursive re-instantiation* of a process allows the specification of infinite, cyclic behavior.

The most important advantages of LOTOS in describing concurrent systems are

Abstraction A small number of concepts (actions, processes, operators, and data) is offered to describe behavior allowing abstraction on different levels and supporting different specification styles [BvLV95]

Compositionality The composition of complex behavior expressions from smaller ones by the use of the LOTOS operators.

Formal Semantics The formal semantics defines an unambiguous model for any LOTOS specification. This gives the opportunity of building tools such as verifier, simulators, or compilers on the base of a common interpretation of LOTOS.

2.2 Functional Behavior of the Steam Boiler System

2.2.1 The Specification Structure

The steam boiler system comprises of a number of physical devices — the steam boiler, a device for measuring the quantity of water in the steam boiler, one pump to provide the steam boiler with water[1], a device for measuring the steam coming out of the steam boiler, a valve to empty the steam boiler —, an operator desk containing the control unit, and a message transmission medium.

In our specification, the functionality of the pump controllers is carried over to the pumps themselves. A pump is able to inform the control unit about its current state and it is able to react properly on the instructions of the control unit. Of course, it is a philosophical question whether to separate the mechanical concerns (the pumps) from the controlling concerns (the pump controllers). Since all other devices such as valve and measurers are also considered as being a device without separate controller, we decided to model the pumps in the same way.

[1] We had to constrain the specification to contain only one pump instead of four pumps in order to restrict the state space and to make the specification automatically analyzable with the LOTOS tools.

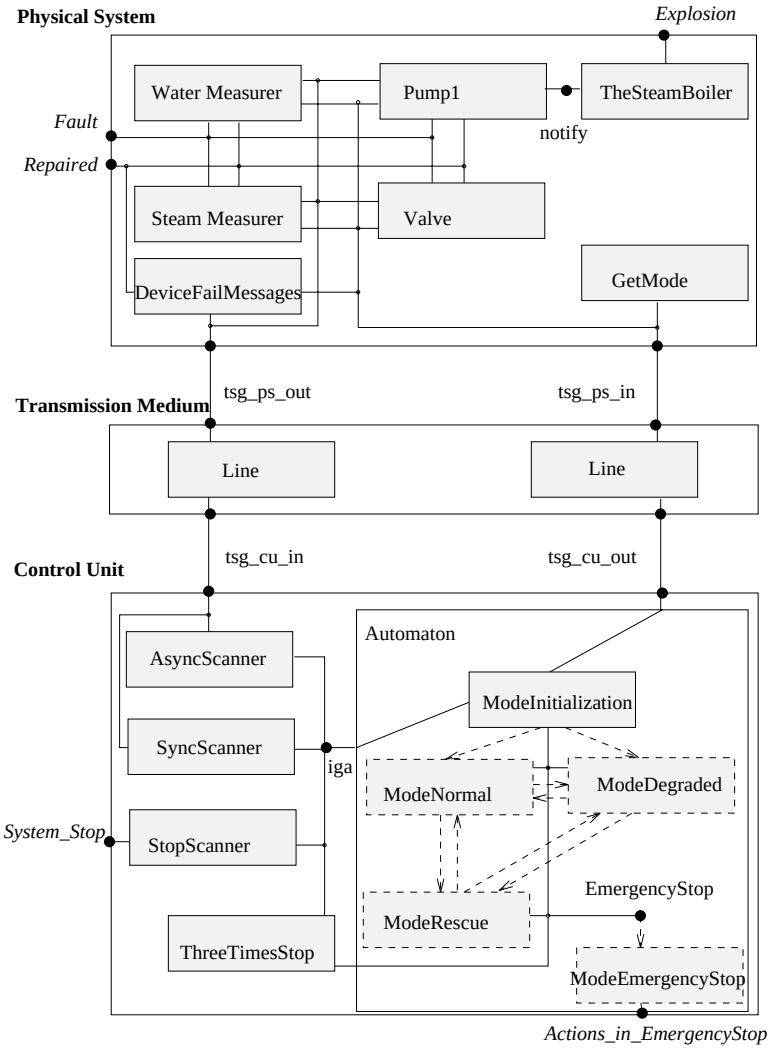

Figure 1: The specification structure

In addition, we assume that device failure messages are received by the operating personnel, which could execute repair actions if needed.

The controller communicates with the physical devices by the use of messages via a transmission medium. The transmission medium connects each physical device with the controller. The transmission time is neglected. The controller works periodically with each cycle consisting of:

1. receiving messages from the physical devices,
2. determining appropriate reaction according to the received information, and
3. transmitting messages to the physical devices.

The specification structure is represented in Fig. 1. The external interface of the steam boiler system, which are gates written in italics, represents the possible inputs and outputs at the physical system and at the operators desk:

1. Interface of the physical system:
   - `Fault`: a physical device is faulty,
   - `Repaired`: the operator has repaired a physical device, and
   - `Explosion`: the steam boiler explodes due to the violation of safety requirements, e.g. the water level exceeds the upper bound for more than 5 sec.
2. Interface of the operators desk:
   - `System_Stop`: the operator can stop the system at any time and
   - `Actions_In_EmergencyStop`: the operator is informed when the control unit enters the emergency stop mode, so that he can execute all necessary actions in order to repair the whole system.

The specification contains three main processes:

1. the physical system (`PS`) that consists of the physical devices, i.e. the valve, the pumps, the steam boiler and the measurers for steam and water, and of two control processes for getting the mode of the control unit and for handling the device failure messages,
2. the transmission medium (`TM`) that consists of two unidirectional unreliable transmission lines for transferring messages from the physical system to the control unit and vice versa, and
3. the control unit (`CU`) that consists of three scanning processes to handle incoming messages from the transmission medium and from outside and of the automaton with separate processes for each mode.

In addition to the external gates, the specification uses internal gates for the communication between processes:

1. Internal gates in the physical system:
   - `notify`: communication between the valve and the steam boiler to take care of the real water inflow.
2. Internal gates in the control unit:
   - `iga`: internal communication between the scanning processes and the automaton, and
   - `EmergencyStop`: internal action that is enabled when a critical situation is reached. This action enforces the emergency stop mode.
3. Internal gates to the transmission medium:
   - `tsg_ps_in`: input port of the physical system,
   - `tsg_ps_out`: output port of the physical system,
   - `tsg_cu_in`: input port of the control unit, and
   - `tsg_cu_out`: output port of the control unit.

The system is represented by the parallel composition in Fig 2. The external and internal gates are defined in line 2 and 6, resp. Line 7-9 define the physical system and the control unit to be independent of each other, except of their common communication over the transmission medium (line 10-11).

```
specification SteamBoiler_System
[Fault,Repaired,ShowEmergencyStop,Actions_In_EmergencyStop,System_Stop,Explosion]:
noexit
...
behaviour
hide tsg_ps_in,tsg_ps_out,tsg_cu_in,tsg_cu_out in
 ( PS [tsg_ps_in, tsg_ps_out,Fault,Repaired,ShowEmergencyStop,Explosion]
   |||
   CU [tsg_cu_in,tsg_cu_out,Actions_In_EmergencyStop,System_stop] )
 |[tsg_ps_in,tsg_ps_out,tsg_cu_in,tsg_cu_out]|
 TM[tsg_ps_in,tsg_ps_out,tsg_cu_in,tsg_cu_out]
where ... endspec
```

Figure 2: The System Behavior

```
process PS [tsg_in,tsg_out,Fault,Repaired,Explosion]: noexit :=
 tsg_out !SteamBoiler!STEAM_BOILER_WAITING !(Null);
 ( ( GetMode [tsg_in]
     |||
     Valve [Fault, Repaired, tsg_in]
     |||
     Measurer [Fault, Repaired, tsg_in, tsg_out](Steam)
     |||
     Measurer [Fault, Repaired, tsg_in, tsg_out](Water)
     |||
     ( Pump [Fault, Repaired, tsg_in, tsg_out, notify] (Pump1)
       |[notify]|
       TheSteamboiler[notify, Explosion](...) ) )
   |||
   ( tsg_in !Controller !PROGRAM_READY !(Null);
     ( DeviceFailMsgs [Repaired, tsg_in, tsg_out] (Pump1)
       |||
       DeviceFailMsgs [Repaired, tsg_in, tsg_out] (Steam)
       |||
       DeviceFailMsgs [Repaired, tsg_in, tsg_out] (Water)
       |||
       tsg_out !SteamBoiler!PHYSICAL_UNITS_READY !(Null); stop ) )
  [>
  ( tsg_in !Controller !MODE !ModeEmergencyStop; stop ) )
where ... endproc
```

Figure 3: The Physical System

2.2.2 The Physical System

The specification of the physical system is given in Fig. 3. The physical system PS has an input and output port, the gates for device faults and repairments, and the gate of explosion (line 1). When the startup message STEAM_BOILER_WAITING is sent to the control unit (line 2), all physical

devices and the GetMode process are instantiated independently of each other (line 4-15).

The pump communicates with the steam boiler in the internal gate notify in order to inform the steam boiler about the real opening or closing of the pump (line 12-15). Using this information, the steam boiler can evaluate the real level of water.

In parallel, the physical system waits for the PROGRAM_READY message from the control unit (line 17). Once this messages is received, the processes for handling failure detection messages from the control unit are instantiated (line 18-22). In addition, the PHYSICAL_UNITS_READY message is sent to the control unit, after which it can leave the initialisation mode (line 24). Whenever a ModeEmergencyStop message is received from the control unit, the whole physical system is interrupted and stops its behavior completely (line 26).

2.2.3 The Control Unit

```
   process CU [tsg_in, tsg_out, Actions_In_EmergencyStop, System_stop]: noexit :=
2    hide  EmergencyStop, Passive_EmergencyStop in
     ( ( hide iga in
4        ( AsyncScanner [EmergencyStop, tsg_in, iga]
           |||
6          StopScanner [EmergencyStop, System_stop, iga]
           |||
8          ( iga !SteamBoiler !PHYSICAL_UNITS_READY !(Null){0};
             SyncScanner [EmergencyStop, tsg_in, tsg_out, iga] )
10       )
         |[iga]|
12       ( tsg_in !SteamBoiler !STEAM_BOILER_WAITING !(Null);
           ( ModeInitialization[EmergencyStop, tsg_in, tsg_out, iga](...)
14           |||
             ThreeTimesStop [EmergencyStop, iga] ) ) )
16     [> Passive_EmergencyStop; stop )
     |[EmergencyStop, Passive_EmergencyStop]|
18   ( EmergencyStop; Passive_EmergencyStop;
       tsg_out !Controller !MODE !ModeEmergencyStop;
20     ModeEmergencyStop[Actions_In_EmergencyStop] )
   where...endproc (*CU*)
```

Figure 4: The Control Unit

Fig. 4 gives the specification of the control unit. It has an input and output port. The Actions_In_EmergencyStop gate informs about the emergency stop mode, while System_stop is used to stop the system externally (line 1).

The internal action EmergencyStop is used by the control unit components to inform the others about an exceptional situation, i.e. about the violation of a safety requirement (line 2). The internal action Passive_EmergencyStop is only needed technically for the interruption of all control unit components once the EmergencyStop has been issued by one of the components (line 2, 16, 18). The interruption is caused by the disabling expression (line 16), which allows us to specify exception handling elegantly.

The control unit CU works in a cyclic manner. This cycle takes place every 5 sec[2] and consists

[2] In the functional specification the timing constraints are not explicitly specified.

of acquiring the measurement values (i.e. level of steam, level of water, pump states), processing the values, and if necessary, to open or close the pumps.

Roughly speaking, gathering all incoming messages is executed by the scanning processes, while the automaton of the control unit is responsible for the decision making according to the current situation.

The `SyncScanner` is responsible for getting all synchronous messages from the physical system (line 9). This process can only work when the physical system is ready, i.e. it is instantiated after the `PHYSICAL_UNITS_READY` message (line 8).

Besides the synchronous cycle, `CU` asynchronously exchanges messages with the physical system. Asynchronous messages are caused by a detected failure of a physical device. Asynchronous messages are scanned by the `AsyncScanner` process (line 4).

The `StopScanner` takes care of the external gate `System_stop`.

Once all incoming messages of the control unit are scanned, they are offered at the internal communication gate `iga` for further processing (line 11). If none of the control unit components is able to process a message that is offered at `iga` within one cycle, a failure situation is assumed and `EmergencyStop` is issued. This ensures that any unawaited external communication leads to the emergency stop mode.

The control units contains five processes — `ModeInitialization`, `ModeNormal`, `ModeDegraded`, `ModeRescue`, `ModeEmergencyStop`, which are also graphically represented in Fig. 1. At any time, only one of them is active and represents the current mode of the control unit. If the current mode of the control unit changes, the current mode process instantiates one of the others. Initially, the control unit is in `ModeInitialization` (line 13). This process is instantiated when the physical system awaits the control unit, i.e. when it issues the `STEAM_BOILER_WAITING` message (line 12).

The mode processes define the automaton of the steam boiler system and constitute the main behavior of the control unit. They decide on proper reactions on incoming synchronous and asynchronous messages (for further details see below).

Finally, the control unit contains a process to observe the occurrence of `System_Stop` from outside in three subsequent periods. This is the `ThreeTimesStop` process (line 15).

2.2.4 The Transmission Medium

```
process TM[tsg_ps_in, tsg_ps_out, tsg_cu_in,tsg_cu_out]: noexit:=
 Line[tsg_ps_out,tsg_cu_in] ||| Line[tsg_cu_out,tsg_ps_in]
 where
 process Line[a,b]: noexit:=
  a ?d: TDevice ?t: TMessage ?p: TParameter;
  (i; b !d !t !p; Line[a,b] (* correct transfer *)
   []
   i; b ?d: TDevice ?t: TMessage ?p: TParameter; Line[a,b] (* data generation *)
   []
   i; Line[a,b] (* loss of data *))
 endproc
endproc
```

Figure 5: The Transmission Medium

The bidirectional transmission medium `TM` that is specified in Fig. 5, consists of two dedicated unidirectional lines `Line` between the physical system and the control unit and vice versa. Each

message that is transmitted contains the physical device (a value of `TDevice`), a specific message type (a value of `TMessage`), and possibly a certain value (a value of `TParameter`) [3].

For example, "`tsg_ps !Water !LEVEL !4`" represents a message that is sent by the physical system, in particular by the device for measuring the level of water. The actual level of water is 4.

The transmission medium may properly transfer (line 6), generate (line 8)[4] or loose messages (line 10), what is specified as nondeterministic choices with the internal action `i`.

2.2.5 Some Purely Functional Processes

This section offers the specification of some processes in the physical system that have a purely functional behavior, i.e. without any time constraints.

```
   process GetMode [tsg_in]: noexit:=
2    tsg_in !Controller !MODE ?p: TParameter [p /= ModeEmergencyStop];
     GetMode[tsg_in]
4  endproc (*GetMode*)
```

Figure 6: The Get Mode Process

- `GetMode` in Fig. 6 is the process to accept all `MODE` messages from the control unit. However, the informal specification says nothing what to do with the mode information in the physical system. Hence, `GetMode` only accepts any `MODE` message (line 2) and recursively re-instantiates (line 3).

```
   process DeviceFailMsgs [Repaired, tsg_in, tsg_out](d: TDevice):noexit :=
2    tsg_in !d !FAILURE_DETECTION  !0 of TParameter;
     tsg_out !d !FAILURE_ACK !0 of TParameter;
4    Repaired !d;
     tsg_out !d !REPAIRED !0 of TParameter;
6    tsg_in !d !REPAIRED_ACK !0 of TParameter;
     DeviceFailMsgs [Repaired, tsg_in, tsg_out](d)
8  endproc (*DeviceFailMsgs*)
```

Figure 7: The Device Failure Process

- The process `DeviceFailMsgs` that is specified in Fig. 7, is responsible for the message handling in the case of device failure. A defective device restrains from doing anything. Hence, the message exchange about failure detection, repairment, and the respective acknowledgements is executed by another process. For any physical device exists one `DeviceFailMsgs` process.

  Whenever the control unit detects a device failure, it sends a `FAILURE_DETECTION` message to the physical system. After receiving this message (line 2), `DeviceFailMsgs` issues an

[3] All data types are explained in Section 4

[4] A random data generation in LOTOS is expressed as a communication between two actions, where both of them request a value of a given type, e.g. `a ?x: Int; stop ||[a]|| a ?y: Int; stop` generates a random integer value.

acknowledgement message, i.e. FAILURE_ACK. Once the device has been repaired by the operator (line 4), a REPAIRED message is send to the control unit (line 5), which answers with REPAIRED_ACK (line 6). This finishes the message handling for a device failure and the DeviceFailMsgs is re-instantiated (line 7).

```
process Measurer [Fault, Repaired, tsg_in, tsg_out](d: TDevice) : noexit :=
  (  tsg_in !d !LEVEL_REQ !(Null);
     tsg_out !d !LEVEL ?l: TParameter [(IsTypeFloatFloat l)]; exit (* generating a random value *)
   [>
     Fault !d;
     Repaired !d; exit )
  >> Measurer [Fault, Repaired, tsg_in, tsg_out](d)
endproc (*Measurer*)
```

Figure 8: The Measurer

- Last but not least, let us discuss the specification of a measurer that gives the level of water or the level of steam in the physical system, which is given in Fig. 8[5]. Whenever the control unit requests the current level with a LEVEL_REQ message (line 2), the measure answers with a random value (line 3)[6]. This reflects the possibility to send completely wrong values to the control unit. This is the reason why the control unit estimates the expected values (see below).

  The measurer stops to send level messages when it is faulty (line 4,5). Only after the repairment it can send measurement values again (line 6,7)

2.3 Functional Verification

The formal semantics of LOTOS assigns to every syntactically and static semantically correct specification a well-defined, unambiguous model, which is a class of structured labelled transition systems. A structured labelled transition system (SLTS) can be seen as an n-ary tree with nodes that are marked with the current behavior expression, and with directed edges that are marked with the occurring action. In addition, an action may be structured by offering a number of data values. Every single path in a SLTS represents a possible execution of the specified system. An SLTS can represent a finite or infinite state space. Liveness and safety properties for finite state systems can be verified, while that is in general not possible for infinite state systems.

2.3.1 The Simulation

We applied LITE — the LOTOS Integrated Tool Environment [pE93] — to validate the steam boiler specification. After checking the syntax and static semantics of the steam boiler specification, the simulator SMILE [Eer94] was used to validate the specification. The simulator offers means

- to execute the specification interactively, i.e. the simulator determines all possible actions in a given behavior expression, under which one of them is chosen for further execution
- to generate automatically all possible executions of a given specification, and

[5] Let us note that the LEVEL message is both used to give the level of water and the level of steam. The messages are differentiate with the TDevice parameter, i.e. Steam or Water.

[6] Please refer to Section 4 for the specification of data types.

- to generate automatically the underlying extended finite state machine[7]

The simulator was very helpful in removing errors in the first versions of the specification. However, the computation time for the generation of all executions and of the extended finite state machines increased exponentially with the size of the specification, what is caused by the interleaving semantics of LOTOS and the state space explosion problem.

2.3.2 The Verification

After validating the specification, we applied CADP — the Caesar/Aldebaran Distribution Package [Gar95] — for verifying the functional correctness. We had to restrict the specification in its size due to restrictions that were imposed by CADP. Therefore, we considered a specification of the steam boiler system where no faults of physical devices can occur.

CADP uses a full state space exploration in order to generate the extended finite state machine. In this sense, it is similar to SMILE, however it is more powerful with respect to the size of the specification that can be analysed. In addition, it is capable to minimise the generated extended finite state machine and to automatically detect deadlocks.

The good news of the verification with CADP is that only one deadlock has been detected. This is exactly the deadlock, when the control unit enters the emergency stop mode.

The bad news are that the verification took several days and that an extended finite state machine with more than 1000 states is hard to proof read.

2.3.3 The Testing Approach

Therefore, we decided to use in addition a testing approach. This allows us to define the requirements on the steam boiler system as separate LOTOS behavior expressions and to verify whether they are met by the specification or not.

A tester defines the requirements to be tested as a LOTOS behavior expression, which possibly ends with a dedicated action, the so called `Success` action. The tester process is composed in parallel with system to be tested and synchronises in all externally visible gates of the system except of the `Success` gate:

```
Tester[<gates>, Success] |[<gates>]| System[<gates>]   (1)
```

The parallel composition (1) of the `Tester` with the `System` to be tested enforces the `System`

- to execute only those behaviors that are defined by the `Tester` and `Success` is reached,
- to end in a deadlock, i.e. the `System` does not contain the specified test sequence and `Success` is not reached, or
- to end in a livelock, i.e. the `System` ends in an infinite cycle with internal actions and `Success` is not reached.

If one considers all executions of the parallel composition (1), it is possible to distinguish three cases:

- The `Success` action does not occur at all, i.e. the requirement is not fulfilled by the system. The system *fails* the test.
- The `Success` action does not occur at least in one execution of the system and does occur at least once. Hence, the requirement may be met, however there are executions which do not adhere to the requirement. The system *may pass* the test.

[7] The LOTOS specification of the steam boiler system has a finite state space.

- The `Success` action does occur in all executions of the system, i.e. the requirement is met. The system *must pass* the test.

All requirements for the steam boiler system can be defined as separate tests. Once, the specification of the steam boiler system passes these tests, the confidence on the specification correctness increases. Since tests reflects only a subset of requirements, which should of course be sufficiently large and should reflect all essential requirements, care has to be taken by the selection of the requirements.

Two important classes of tests are *acceptance* and *rejection* tests. In the first case, the system is tested to accept a given sequence of interactions with the environment. In the latter case, the system is tested to reject sequences of interactions that are not allowed to occur.

Safety requirements can be represented by must acceptance tests or by rejection tests. Liveness requirements are reflected by may acceptance tests.

In order to prove the correctness of the steam boiler specification, we adopted the method of testing described above.

Since the steam boiler system has to enter the emergency stop mode under all faulty conditions, we only had to use acceptance tests. In addition to the tests on the external behavior we applied tests on the internal correctness. For this purpose, the tester listens also to the internal transmission medium at the `tsg_ps` and `tsg_cu` gates.

Subsequently, we consider some of the functional tests that we used to check the correctness of the steam boiler specification. We give only the main part of the test in order to ease the understanding, for example most of the tests require that the control unit has finished the initialisation mode. The kind of test is given in parenthesis.

- These two tests check whether a proper start up of the steam boiler system (may accept) is possible:

```
tsg_ps_out !SteamBoiler !STEAM_BOILER_WAITING  !(Null);
tsg_ps_out !Steam !LEVEL !(Null);
tsg_ps_out !Water !LEVEL !(Null);
tsg_ps_out !SteamBoiler !PHYSICAL_UNITS_READY  !(Null);
Success; stop

tsg_cu_out !Controller  !MODE  !ModeInitialization ;
tsg_cu_out !Steam  !LEVEL_REQ  !(Null);
tsg_cu_out !Water  !LEVEL_REQ  !(Null);
tsg_cu_out !Controller  !MODE  !ModeInitialization;
tsg_cu_out !Controller  !PROGRAM_READY  !(Null);
Success; stop
```

- The message `STEAM_BOILER_WAITING` is awaited by the control unit only once (must accept).

```
tsg_ps_out !SteamBoiler !STEAM_BOILER_WAITING;
tsg_ps_out !SteamBoiler !STEAM_BOILER_WAITING;
Actions_In_EmergencyStop {period}; Success; stop
```

- The message `PHYSICAL_UNITS_READY` is awaited by the control unit only once (must accept).

```
tsg_ps_out !SteamBoiler !PHYSICAL_UNITS_READY;
tsg_ps_out !SteamBoiler !PHYSICAL_UNITS_READY;
Actions_In_EmergencyStop {period}; Success; stop
```

Name	Syntax	Explanation
Time-constrained action prefix	$g\, o_1 \ldots o_n\, \{t\, in\, T\}\, [SP]; Q$	Observable action g with value offering within time window T (the life reducer) and afterwards behavior Q
@-operator in an action prefix	$g\, o_1 \ldots o_n\, @t; Q$	After the occurrence of g the time variable t contains the time between action enabling and action occurrence
Time-constrained internal action prefix	$i\{t\, in\, T\}; Q$	Internal action i within time window T and afterwards behavior Q
Delay	$Wait(d); Q$	Wait time d and afterwards behavior Q

Table 2: Timed Operators in TE-LOTOS

- The message `STEAM_FAILURE_DETECTED` from the control unit is correctly acknowledged by the physical system (may accept).

```
tsg_ps_in !Steam !FAILURE_DETECTED !0;
tsg_ps_out !Steam !FAILURE_ACK;
Success; stop
```

3 The Time-Extended Specification in TE-LOTOS

3.1 Overview on Time-Extended LOTOS

With the advent of high-speed networks and new communication services with their stringent QoS requirements, the need to formally capture real-time constraints gained in importance. Numerous time-extensions of LOTOS have been defined only recently, while actually the two proposals [LL94] and [QMFL94] for time enhancements are converging to timed extended LOTOS (TE-LOTOS [LLdFQ95]) as the main candidate for the upcoming LOTOS revision by ISO.

Time Extended LOTOS has a discrete or dense time domain. A few new operators are introduced for describing timing constraints. Life reducers define a time window for an action, so that any synchronization within that action is bounded in time. If the synchronization does not occur in time, the behavior stops even if the environment offers this gate some time later. A delay operator can be used to describe the passage of time.

Please refer to Table 2 for a list of operators of TE-LOTOS that will be used in the steam boiler specification. The advantages of LOTOS in describing concurrent systems carry over to TE-LOTOS in describing real-time systems.

3.2 The Timed Behavior of the Steam Boiler System

3.2.1 The Pumps

```
process Pump[Fault, Repaired, tsg] (d: TDevice): noexit:=
 hide req_open, req_close, notify in
 ( pmp [req_open, req_close, notify, tsg] (d, PumpIsClosed)
  |[req_open, req_close, notify]|
   pumpswitch [req_open, req_close, notify] )
 [>
 ( Fault !d; Repaired !d; Pump [Fault, Repaired, tsg] (d)
 where
 process pmp[req_open, req_close, notify, tsg]
            (d: TDevice, state : TPumpState): noexit:=
    ( tsg !d !OPEN_PUMP; req_open;
      pmp [req_open, req_close, notify, tsg] (d, state) )
 [] ( tsg !d !CLOSE_PUMP; req_close; notify ! PumpIsClosed {0};
      pmp [req_open, req_close, notify, tsg] (d, PumpIsClosed) )
 [] ( notify ?new_state: TPumpState;
      pmp [req_open, req_close, notify, tsg] (d, new_state) )
 [] ( tsg !d !PUMP_STATE !state;
      pmp [req_open, req_close, notify, tsg] (d, state) )
 endproc (*pmp*)
 process pumpswitch [req_open, req_close, notify]:noexit :=
  ( ( req_open; Ticker[req_open, req_close, notify](0) )
    [>
    ( req_close; notify !PumpIsClosed; exit ) )
  >> pumpswitch[req_open, req_close, notify]
 endproc (*pumpswitch*)
 process Ticker[req_open, req_close, notify](eps: Time): exit:=
    Wait(switchtime-eps); notify ! PumpIsOpen; exit
 [] req_open{t}; Ticker[req_open, req_close, notify](eps+t)
 endproc (*ticker*)
endproc (*Pump*)
```

Figure 9: The Pumps

We discuss the specification of the timed behavior of the steam boiler system in TE-LOTOS exemplarily for those processes with severe timing constraints. Firstly, we discuss the specification of a pump (given in Fig. 9) as a representative of the physical system. A pump is identified by its device identifier (a value of type `TDevice`)[8]. The rules for its behavior are the following:

- A pump is either open or closed (the initial state). The state information is offered to the control unit whenever needed without any delay (line 17).
- A pump can handle `OPEN_PUMP` and `CLOSE_PUMP` messages, where a closed pump ignores `CLOSE_PUMP` and an open pump ignores `OPEN_PUMP`.
- An open pump is closed immediately after receiving `CLOSE_PUMP` (line 13).

[8] According to our restriction, there is only one pump identified by `Pump1`.

- A closed pump is opened within 5 sec after receiving `OPEN_PUMP`. An additional `OPEN_PUMP` message arriving within this time interval has no effect. A `CLOSE_PUMP` message arriving within this time makes the `OPEN_PUMP` message ineffective, i.e. the pump remains closed (line 11, 20-25).
- A pump can break down and stay completely inactive until it is repaired (line 7).

The subprocess `pmp` is responsible for the message exchange with the control unit. The subprocess `pumpswitch` specifies the behavior to open a closed pump. During that period additional open and close pump requests are possible. The `Ticker` counts the time interval of 5 sec.

3.2.2 The Scanners

```
   process SyncScanner [EmergencyStop, tsg_in, tsg_out, iga] : noexit :=
2    tsg_out !Steam !LEVEL_REQ !(Null) {0};
     ( tsg_in !Steam !LEVEL ?qs: TParameter {0};
4      iga !Steam !LEVEL !qs;
       tsg_out !Water !LEVEL_REQ !(Null) {0};
6      ( tsg_in !Water !LEVEL ?qw : TParameter {0};
         iga !Water !LEVEL !qw;
8        tsg_out !Pump1 !PUMP_STATE_REQ !(Null) {0};
         ( tsg_in !Pump1 !PUMP_STATE ?ps1 : TPumpState {0};
10         iga !Pump1 !PUMP_STATE !ps1;
           Wait(5)  (* we continue in next cycle *)
12         SyncScanner [EmergencyStop, tsg_in, tsg_out, iga]
           []
14         Wait(epsilon); EmergencyStop; stop )
         []
16       Wait(epsilon); EmergencyStop; stop )
       []
18     Wait(epsilon); EmergencyStop; stop )
   endproc (*SyncScanner*)
```

Figure 10: The Sychronous Scanner

The process `Sync_Scanner` that is given in Fig. 10 scans every 5 sec the state information of physical devices such as the water level, the steam level, and the pump states. Scanning messages is realized by a pair of request and answer message. For example, `tsg_out !Steam !LEVEL_REQ !(Null) 0` requests the level of steam from the steam measurer (line 2) and awaits the answer immediately with `tsg_in !Steam !LEVEL ?qs: TParameter 0` (line 3). Afterwards, the information is offered at the internal gate `iga` to the mode processes, immediately (line 4). Whenever the current mode process is not able to take over the information, i.e. an internal error in the control unit occurred, `EmergencyStop` is issued after time `epsilon` (line 17-18). `Epsilon` is small enough to react on this exception properly [9]. Once all information is scanned, `Sync_Scanner` waits one period (line 11) before the next scan cycle, what is specified as a re-instantiation of the process (line 12).

[9] One shortcoming of TE-LOTOS becomes obvious here: the lack of priorities in a nondeterministic choice expression makes it impossible to decide instantaneously whether the `Worker` is ready or not. Instead, we have to use the small delay `epsilon` before issuing the timeout.

```
process AsyncScanner [EmergencyStop, tsg_in, iga]:noexit :=
  tsg_in ?d: TDevice ?msg: TMessage ?p: TParameter
  [(d ne Controller) and ((msg eq REPAIRED_ACK) or (msg eq FAILURE_ACK))];
  ( iga !d !msg !p {5}; AsyncScanner [EmergencyStop, tsg_in, iga]
    []
    Wait (period+epsilon); EmergencyStop; stop )
endproc (*AsyncScanner*)
```

Figure 11: The Asychronous Scanner

The asynchronous scanner `AsyncScanner` is specified in Fig. 11. It is able to receive a message from the physical system at any time (line 2). It only considers messages from the physical devices, which do not belong to the synchronous ones (line 3). It offers the received message at the internal gate `iga` of the control unit (line 4). If no process in `CU` is awaiting this message within one cycle, `EmergencyStop` is initiated (line 6).

```
process StopScanner [EmergencyStop, System_stop, iga] :noexit :=
  System_stop;
  ( iga !Controller !SYSTEM_STOP  !(Null) {5};
    StopScanner [EmergencyStop, System_stop, iga]
    []
    Wait (period+epsilon); EmergencyStop; stop )
endproc (*StopScanner*)
process ThreeTimesStop [EmergencyStop, iga]:noexit :=
  iga !Controller !SYSTEM_STOP !(Null);
  ( iga !Controller !SYSTEM_STOP  !(Null){period};
    ( iga !Controller !SYSTEM_STOP  !(Null){period};
      EmergencyStop; stop
      []
      Wait(period); ThreeTimesStop [EmergencyStop, iga] )
    []
    Wait(period); ThreeTimesStop [EmergencyStop, iga] )
endproc (*ThreeTimesStop*)
```

Figure 12: The Stop Scanners

The processes that observe three subsequent `System_Stop` messages from the operator are specified in Fig. 12. The scanner for the `System_Stop` messages from outside is able to receive the `System_Stop` messages at any time. `ThreeTimesStop` issues an `EmergencyStop` when receiving three times within (at most) three subsequent periods.

3.3 Timed Verification

Due to the lack of tools for TE-LOTOS we applied manually tests to the steam boiler specification. Four of them are given below.

- This test checks the important safety requirement that after three times `System_Stop` in consecutive periods, the steam boiler has to enter the `EmergencyStop` mode (must accept).

```
System_Stop{period}; System_Stop{period}; System_Stop{period};
Actions_In_EmergencyStop{0}; Success; stop
```

- In the case that the device for measuring the level of water and the device for measuring the level of steam are defective or one of them is faulty, the `CU` has to enter the `EmergencyStop` mode within 5 sec (must accept).

```
(tsg_cu_out !Steam !FAILURE_DETECTION !0;
 tsg_cu_out !Water !FAILURE_DETECTION !0; exit
 []
 tsg_cu_out !Water !FAILURE_DETECTION !0;
 tsg_cu_out !Steam !FAILURE_DETECTION !0; exit)
>> Actions_In_EmergencyStop{period}; Success; stop

(Fault !Steam ; exit
 []
 Fault !Water; exit)
>> Actions_In_EmergencyStop{period}; Success; stop
```

- the measurement values for the level of water are received each 5 sec or the emergency stop mode is reached (must accept):

```
tsg_cu_in !Water !Level !x [(isTypeFloat x)];
tsg_cu_in !Water !Level !y [(isTypeFloat y)] @t [t /= 5];
(EmergencyStop; Success; stop
 []
 i; stop)
```

4 The Data-Extended Specification in TE-LOTOS with Gofer

4.1 Overview on Gofer

The data type part of LOTOS — ACT ONE — has several problems with specifications of practical relevance. It uses equational specifications and initial algebra semantics. Some of the major drawbacks of equational data definitions are

- All operations have to be defined in a total manner. Partiality has to be dealt with explicitly. This usually results in long-winded and verbose specifications.
- Equational specifications are semi-decidable. There is no general algorithm to decide the equality of two values. In order to have a tool support, usually some kind of functional interpretation such as directioning equations from left to right is used. However, this technique yields another semantics and is only applicable to a restricted class of data type specifications.
- In several cases it is not easy to find a complete and correct set of equations.

Hence, we decided to use the functional data types offered by Gofer for the description of data dependencies instead. Most importantly, functional specifications are executable and concise. They offer the basic concepts of algebraic data type languages such as abstract data types, overloading, and parameterisation. The syntax leads to short and concise specifications, using widely the well-known mathematical notation.

The functional programming language *Gofer* [Jon93] — a derivative of Haskell [HJea92] — is a non-strict, strongly typed and purely functional language with lazy evaluation. It offers, among a lot of other features, a polymorphic static type system with algebraic data types, pattern matching, and infix and prefix operators.

A number of standard data types such as Integer, Float, Char, Strings, and functional types are predefined. New types can be defined as type synonyms or as (new) algebraic data type.

With a type synonym one declares just another name for an existing type. The type checker is not able to distinguish between these two types:

```
type TQoS = Float
```

Algebraic data types can be declared as enumeration types and/or recursive types. Recursive types are not used in the steam boiler specification. An example for an enumeration type is

```
data TMode =   ModeInitialization | ModeNormal | ModeEmergencyStop
             | ModeRescue | ModeDegraded
```

Gofer supports two types of polymorphism:

1. With *parametric polymorphism* it is possible to use type variables in the definition of an algebraic data type. To build a binary tree whose leaves are of any type (but all the same), one can write:

   ```
   data Tree a = Leaf a | Branch (Tree a) (Tree a)
   ```

   The identifier `a` is a type variable, which can be substituted by any concrete type.

2. Gofer offers *ad-hoc-polymorphism* by so-called type-classes: a type class is a set of types with the same operations on all types, i.e. with the same syntax but possibly with different implementations.

   For example, the common operations on type class `Num` are `+, -, *` and `/`. The predefined types `Int` and `Float` are instances of this type class. New types can be declared as members of a given type class and "inherit" the operations of their type class. A type instance can belong to more than one type class.

   Some important type classes are `Eq` — implementing equality and non-equality — and `Ord` — partial ordering.

Gofer is a strongly typed language, i.e. an operation may only be used, if the inferred types for the actual and formal parameters of the operation are the same or if they (in the case of polymorphic functions or types) can be matched. The type of a given expression is inferred by the use of context information and is determined as the most general type that is possible[10].

We use Gofer for the description of data dependencies due to its powerful features, compact notation, easy understanding, and the existence of a well-defined semantics[11].

[10] This kind of type system was first used in ML, based on work of Milner [Mil78]

[11] In TE-LOTOS, we restricted the use of functional values. Large overhead would be needed when using functional values (lambda-abstractions or partial function applications) for an information exchange in synchronising gates. This is due to the fact that the set of free variables used by the function must be transferred too. Hence, we excluded such cases.

4.2 Data Dependencies of the Steam Boiler System

4.2.1 The Data Types of the Steam Boiler System

Most of the data types in the steam boiler specifications are enumerations which represent messages between system components. These enumerations are of class `Eq`, so that their elements are comparable via the '`==`' and '`/=`' operations:

```
class Eq where
  (==), (/=) :: a -> a -A Bool
  x /= y = not (x == y)
```

With the given definition of type class `Eq` it is possible to derive the implementation of `/=` from that of `==`.

Every instance of `Eq` has to define a total function, which decides for every pair of values, if they are equal or not. In the instance declaration, this function is associated with the operation `==`[12]:

```
instance Eq TPumpState where
    PumpIsOpen     ==  PumpIsOpen       =  True
    PumpIsClosed   ==  PumpIsClosed     =  True
    _              ==  _                =  False
```

In the steam boiler specification we make use of the following types:

TMessage : This enumeration type contains all messages that are exchanged between the control unit and the physical system

```
data TMessage =    LEVEL | LEVEL_REQ| PUMP_STATE | PUMP_STATE_REQ
                 | MODE | PHYSICAL_UNITS_READY
                 | STEAM_BOILER_WAITING | FAILURE_DETECTION | FAILURE_ACK
                 | REPAIRED | REPAIRED_ACK | TVALVE | OPEN_PUMP
                 | CLOSE_PUMP | SYSTEM_STOP | PROGRAM_READY
```

TDevice : This enumeration type contains all physical devices.

```
data TDevice =    Steam | Water | Valve | Pump1 | SteamBoiler | Controller
```

TPumpState : This type describes the set of possible states of a pump.

```
data TPumpState = PumpIsOpen | PumpIsClosed
```

TPumpAction : This type defines all things that can be done with a pump.

```
data TPumpAction = OpenPump | ClosePump | DoNothing
```

TMode : This enumeration type identifies the operational modes of the control unit. The mode is transmitted to the physical system in every cycle.

```
data TMode =    ModeInitialization | ModeNormal | ModeEmergencyStop
              | ModeRescue | ModeDegraded
```

[12] The rule `_ == _ = False` matches only if none of the textually preceding rules matches. That means, that only identical pump states are really equal.

TInitState : This type contains all sub states of the control unit in the initialisation mode

```
data TInitState =    Init_CheckQS | Init_CheckQW | Init_CloseV
                   | Init_CloseP |  Init_PUReady
```

TQoS, TQoW, TQoP : `TQoS, TQoW` are type synonyms of type `Float` and declare the range of possible values transmitted by the devices for measuring the level of water and the level of steam. `TQoP` is a type to represent pump rates. We explicitly make use of the whole range of type `Float` in order to model the transmission of wrong values.

```
type TQoS = Float
type TQoW = Float
type TQoP = Float
```

TParameter : This type is an aggregate type for all possible parameter values transmitted with a message: a `Float` value for the `LEVEL` messages, a value of type `TMode` for the `MODE` message, a value of type `TPumpState` for the states of the pumps and a special `Null` value indicating missing parameters. Additionally, helper functions determine the actual type of a value of `TParameter`. The function `isMode` also checks for a specific `TMode` value.

```
data TParameter =     Null
                    | PFloat           Float
                    | PTMode           TMode
                    | PTPumpState      TPumpState

isMode :: TParameter -> TMode -> Bool
isMode (PTMode x) y  =   x == y
isMode _  _          =   False

isTypeFloat :: TParameter -> Bool
isTypeFloat (PFloat _)  =  True
isTypeFloat _           =  False

isTypeTPumpState :: TParameter -> Bool
isTypeTPumpState (PTPumpState _)  =  True
isTypeTPumpState _                =  False
```

In addition, the special type `Time` is predefined in TE-LOTOS to be discrete or dense and is used to represent delays or durations. We assume `Time` to be a synonym of type `Float` and to be an `instance` of type class `Ord`, i.e. to be partially ordered.

Several Gofer functions are defined, for example to determine the adjusted values for the level of water. This function uses static bounds, i.e. `0 <= q <= C`, and dynamic bounds, i.e. `qc1, qc2`.

```
determ_qa:: TQoW -> TQoW -> TQoW -> TQoW -> TQoW
determ_qa q qc1 qc2 qres  | (q < 0) = qres
                          | (q > C) = qres
                          | (q < qc1) = qres
                          | (q > qc2) = qres
                          | otherwise = q
```

Other functions determine the adjusted level of water, failures of the steam and water measurers, and needed control actions on the pumps.

4.2.2 The Steam Boiler

We now describe some parts of our specification, where data dependencies play the key role for the correct behaviour of the steam boiler system.

We added a process to the physical system that describes the behaviour of the steam boiler itself, called `TheSteamboiler`(Fig. 13). This process allowed us to explicitly model the situation, where safety requirements are violated and the steam boiler explodes. It provides also a good example for the combined use of data and time dependencies.

```
process TheSteamboiler[notify, Explosion]
(currlevel: TQoW, acctime: Time, curr_inflow: TQoP, curr_outflow: TQoS):
exit:=
 let ctime : Time = 0.1 in
 (   [currlevel >= M2] ->
     ( [acctime >= period] -> Explosion; stop
       []
       [acctime < period]  -> exit(acctime) )
  [] [currlevel <= M1] ->
     ( [acctime >= period] -> Explosion; stop
       []
       [acctime < period]  -> exit(acctime) )
  [] [(currlevel > M1) && (currlevel < M2)] ->  exit(0) )
 >>  accept new_acctime: Time in
 (   Wait(ctime);
     TheSteamboiler[notify, Explosion](currlevel + ctime * (curr_inflow - curr_outflow),
                                       (if (new_acctime == 0) then 0 else new_acctime + ctime),
                                       curr_inflow, curr_outflow)
  [] notify !PumpIsOpen @t [curr_inflow < P]
     TheSteamboiler[notify, Explosion](currlevel + t * (curr_inflow - curr_outflow),
                                       (if (new_acctime == 0) then 0 else new_acctime + t),
                                       curr_inflow + P,curr_outflow)
  [] notify !PumpIsClosed @t  [curr_inflow > 0]
     TheSteamboiler[notify, Explosion](currlevel + t * (curr_inflow - curr_outflow),
                                       (if (new_acctime == 0) then 0 else new_acctime + t),
                                       curr_inflow - P, curr_outflow) )
endproc (*TheSteamBoiler*)
```

Figure 13: The Steam Boiler

Its operation is as follows: the steam boiler is filled with water, and its current water level is denoted by *currlevel* (line 2). The pumps, if open, are pouring water with a constant rate into the boiler (inflow) and, on the other side, the steam is leaving the boiler with a constant rate (outflow)[13] (lines 18, 22, 26). Inflow and outflow are given in litres per second. The rate of the inflow changes accordingly to opening or closing the pumps: opening a pump increases the inflow by the fixed amount of P (capacity of the pumps in litres per sec), closing a pump decreases it by the same rate (lines 19-26). The current level of water is calculated every 0.1 sec from the last level of water, the inflow and the outflow (line 15-18). If the relation $currlevel \geq M2$[14] holds for five seconds or more, the boiler explodes, the same condition holds for $currlevel \leq M1$ (lines 6,

[13] Since no outflow discipline was given in the problem statement, we decided to use the simplest one. Another possibility is an outflow proportional to the level of water

[14] The meaning of $M1$ and $M2$ is described in [Abr95]: beyond these bounds the steam boiler is in danger of

10). The variable *acctime* is incremented every time the water level exceeds one bound, or it is set to 0 if the water level is within the bounds[15]. A pump can be opened at any time t between two cycles (so, i.e. $0 \leq t \leq 0.1$; for this point of time the new water level is computed and the inflow is increased by P. Similar rules apply when a pump is closed (lines 19-26).

4.2.3 The Normal Mode Process

One of the most important parts of the control unit specification describes its operation in the normal mode: in the absence of errors the control unit has to maintain the water level between $N1$ and $N2$ (with $M1 < N1 < N2 < M2$) and it must not reach one of the bounds $M1$ or $M2$. For this reason it communicates every five seconds (control unit cycle time) with the physical system in order to acquire the measurements and state information from the pumps and from the devices for measuring the level of outgoing steam and the level of water.

The `ModeNormal` process executes the following computations based on the current measurements (see also [Abr95]). Some of these computations are executed in local processes such as `GetNewState`, `NewDefect`, and `DoWithPump`:

Compute adjusted values Adjusted values are computed in `GetNewState` Fig.15, lines 8-11.

For example, the adjusted values for the level of water: let q be the measurement value gathered from the measurement device and $qc1$ and $qc2$ be the estimated upper and lower bounds for the water level (the estimation taken in the last cycle is based on a simple model of the system dynamics). If $qc1 \leq q \leq qc2$ holds, q is taken as adjusted value for the upper and lower bound of the water level ($qa1$ and $qa2$). Otherwise the measurement device is assumed to be defective and the estimated bounds $qc1$ and $qc2$ are taken for $qa1$ and $qa2$:

```
let qa1 : TQoW = (if dqw_defective then qc1
                                   else (determ_qa q qc1 qc2 qc1)
                 ) in
```

In the same way the measurement values for the level of steam and for the pumps are treated. With these calculations the control unit can determine, whether the physical devices are defective or not. For this we defined the Gofer functions `determ_steamfail` and `determ_levelfail`

Compute the estimated values Estimated values are also computed in `GetNewState` Fig.15, lines 14-17, based on the current adjusted values and the model of the system dynamics.

For example the new estimated values for the level of water:

```
let new_qc1 : TQoW = (qa1 - va2*period - (0.5)*U1*period*period + pa1*period) in
let new_qc2 : TQoW = (qa2 - va1*period + (0.5)*U2*period*period + pa2*period) in
```

Compute the next pump action This is done with a specific Gofer function in process `DoWithPump`:

```
determ_pumpaction :: TQoW -> TQoW -> TPumpAction
determ_pumpaction qa1 qa2    | (qa1 <= N1) && (qa2 <= N1)      =   OpenPump
                             | (qa1 <= N1) && (qa2 > N1) &&
                               (qa2 < N2)                      =   OpenPump
```

exploding.

[15] We assume the water level to be "steady": the pathological case of $currlevel \geq M2$ in this cycle and $currlevel \leq M1$ in the next cycle should never occur

```
| (qa1 <= N1) && (qa2 >= N2)      =   DoNothing
| (qa1 > N1) && (qa1 < N2) &&
  (qa2 > N1) && (qa2 < N2)        =   DoNothing
| (qa1 > N1) && (qa1 < N2) &&
  (qa2 >= N2)                     =   ClosePump
| (qa1 >= N2) && (qa2 >= N2)      =   ClosePump
```

```
process ModeNormal[EmergencyStop, tsg_in, tsg_out, iga]
(dqs_defective,dqw_defective,p1_defective: Bool, vc1, vc2: TQoS, qc1, qc2: TQoW):
noexit:=
    tsg_out !Controller!MODE! (PTMode ModeNormal) {0};
    GetNewState[EmergencyStop, tsg_out, iga]
              (dqs_defective,dqw_defective,p1_defective, vc1, vc2, qc1, qc2)
    >> accept new_dqs_defective,new_dqw_defective,new_p1_defective: Bool,
              qa1, qa2:  TQoW, new_vc1, new_vc2: TQoS, new_qc1, new_qc2: TQoW in
    DoWithPump[EmergencyStop, tsg_out]
              (Pump1, new_p1_defective, (determ_pumpaction qa1 qa2))
    >>
    [new_dqw_defective == True] ->
         ModeRescue[EmergencyStop, tsg_in, tsg_out, iga](...)
    []
    [new_dqw_defective == False] ->
      ( [new_dqs_defective || new_p1_defective ]  ->
                ModeDegraded[EmergencyStop, tsg_in, tsg_out, iga](...)
        []
        [not(new_dqs_defective) && not(new_p1_defective)]  ->
            ModeNormal[EmergencyStop, tsg_in, tsg_out, iga](...) )
endproc (*ModeNormal*)
```

Figure 14: The Mode Normal Process

Besides the computations, the `ModeNormal` process has to execute the following tasks:

- The physical system has to be kept informed about the internal state of the control unit; this information is transmitted using the `MODE` message (see Fig. 14, line 4).
- The control unit must always check the water level: if one of the adjusted values $qa1$ or $qa2$ or one of the estimated values $qc1$ or $qc2$ exceeds one of the bounds $M1$ or $M2$ the control unit has to enter the emergency stop state immediately (Fig. 15, lines 30-35).
- Based on the acquired information on the current state of the physical devices (defective or not) the following mode of the control unit has to be determined (Fig. 14, lines 12-20).

Firstly, we show the process `GetNewState` in Fig 15: it acquires the measurement values, computes the adjusted values and new estimations, determines whether the physical devices are defective or not and checks the safety requirements.

The process `NewDefect` that is specified in Fig. 16, determines whether a given device is currently defective or not. A device is said to be defective if:

1. it is already defective and not yet repaired or

```
process GetNewState[EmergencyStop, tsg_out, iga]
(dqs_defective, dqw_defective, p1_defective: Bool, vc1, vc2 : TQoS, qc1, qc2 : TQoW):
exit(Bool, Bool, Bool, TQoW, TQoW, TPumpState, TQoS, TQoS, TQoW, TQoW):=
    iga !Steam !LEVEL ?v: TParameter [(isTypeFloatFloat v)];
    iga !Water !LEVEL ?q: TParameter [(isTypeFloatFloat q)];
    iga !Pump1 !PUMP_STATE ?ps1: TPumpState [(isTypeTPumpStateTPumpState p1)];
    (
      let va1:TQoS = (if dqs_defective then vc1 else (determ_va v vc1 vc2 vc1)) in
      let va2:TQoS = (if dqs_defective then vc2 else (determ_va v vc1 vc2 vc2)) in
      let qa1:TQoW = (if dqw_defective then qc1 else (determ_qa q qc1 qc2 qc1)) in
      let qa2:TQoW = (if dqw_defective then qc2 else (determ_qa q qc1 qc2 qc2)) in
      let pa1 : TQoP = (if p1_defective then 0 else (determ_pa p1)) in
      let pa2 : TQoP = (if p1_defective then P else (determ_pa p1)) in
      let new_vc1 : TQoS = (va1 - U2 * period) in
      let new_vc2 : TQoS = (va2 + U1 * period) in
      let new_qc1 : TQoW = (qa1 - va2*period - (0.5)*U1*period*period + pa1*period) in
      let new_qc2 : TQoW = (qa2 - va1*period + (0.5)*U2*period*period + pa2*period) in

    ( NewDefect[EmergencyStop, tsg_out, iga](Steam,dqs_defective,(determ_steamfail v vc1 vc2))
      >> accept new_dqs_defective : Bool in
      NewDefect[EmergencyStop, tsg_out, iga](Water,dqw_defective,(determ_levelfail q qc1 qc2))
      >> accept new_dqw_defective : Bool in
      ( [new_dqw_defective && new_dqs_defective]  -> EmergencyStop; stop
        []
        [not (new_dqw_defective) || not (new_dqs_defective)] ->
          NewDefect[EmergencyStop, tsg_out, iga](Pump1, p1_defective)
          >> accept new_p1_defective : Bool in
             exit (new_dqs_defective, new_dqw_defective,new_p1_defective,
                   qa1, qa2, ps1, new_vc1, new_vc2, new_qc1, new_qc2) )
     [> (  (* really dangerous *)
              ([qa1 <= M1]  -> EmergencyStop; stop)
           []  ([qa2 >= M2]  -> EmergencyStop; stop)
           (* probably dangerous *)
           []  ([qc1 <= M1]  -> EmergencyStop; stop)
           []  ([qc2 >= M2]  -> EmergencyStop; stop) )
endproc (*GetNewState*)
```

Figure 15: The Get New State Process

2. it is not defective but it behaves in a strange way (its `defect_condition` is true)

Otherwise it is assumed to be ok.

After acquiring all necessary state information as explained, `ModeNormal` determines what to do with the pumps, given the current adjusted values for the level of water. For this calculation the Gofer function `determ_pumpaction` is used. The process `DoWithPump` is responsible for transmitting the right messages, i.e (`OPEN_PUMP` or `CLOSE_PUMP`, according to the current pump state (defective or not) and the chosen pump action.

```
 process NewDefect [EmergencyStop, tsg_out, iga]
(p: TDevice, p_defective: Bool, defect_condition: Bool):
exit (Bool) :=
      [p_defective == True]    ->
        ( iga !p !REPAIRED !(Null) {0};
          ( tsg_out !p !REPAIRED_ACK !(Null) {0}; exit (False)
            []
            Wait (epsilon); EmergencyStop; stop )
        []
        iga !p !FAILURE_ACK !(Null) {0}; exit (True)
        []
        Wait(epsilon); exit (True) )
      []
      [p_defective == False]   ->
        ( [defect_condition == True] ->
              tsg_out !p !FAILURE_DETECTION !(Null) {0}; exit(True)
              []
              Wait (epsilon); EmergencyStop; stop )
          []
          [defect_condition == False] -> exit (False) )
endproc (*NewDefect*)
```

Figure 16: The New Defect Process

4.3 Verification of Data Dependencies

In the course of verifying the data dependent parts we used the Gofer interpreter and verified the data type declarations and the function definitions.

However, the integrated specification with Gofer data types and TE-LOTOS behavior could only be verified manually due to the lack of tools for TE-LOTOS and for the integration of Gofer data types in a TE-LOTOS specification. On the other hand, after introducing the steam boiler itself with the special external gate `EXPLOSION`, it is very easy to write a test for the basic safety requirements (see below).

However, attempting to develop more specific tests for liveness properties, which include the level of water (e.g. the level of water must not exceed the lower bound $M1$), we encountered a problem: which value should we take as "the level of water"? Since the measurer for the level of water possibly transmits wrong values, these are not reliable. The other possibility is, that the steam boiler-process offers his value for the water level. But this approach has two major drawbacks:

- we have to change the specification of the steam boiler process in order to add an observability feature for specific tests
- the environment of the steam boiler must always be able to synchronise with the steam boiler in time. If the environment fails to do so, wrong results are obtained.

Hence, we restrained from redefining our specification. Therefore, the following test captures only the basic safety requirement; liveness and performance properties are not considered:

After the correct initialisation of the steam boiler system, the control unit always reaches the safe emergency stop state before the steam boiler explodes (must accept):

```
tsg_cu_out !Controller !MODE ?x:TParameter [not (isMode x ModeEmergencyStop)];
(Explosion; stop
 []
 EmergencyStop; Success; stop)
```

5 Answers to the Questionnaire

1. Does the Solution comprise a requirements specification?

   The requirements are explicitly specified as tests. We have tests for the individual system components and for global safety requirements. The functional tests were used to automatically check the correctness of the functional specification by full state space exploration.

   There are special safety tests for:

   - the reaction on three times stop: reaching the safe emergency stop state
   - the distinction between correct and erroneous message transfer, and reaching the safe emergency stop state in case of transmission errors.
   - reaching the safe emergency stop state before the water level exceeds the given upper or lower bound

   Liveness tests were applied to the system components individually. However, the global liveness requirement that while no error occurs, the emergency stop state must not be entered, could not be expressed.

   There are not tests for the performance of the systems since performance issues are not considered.

2. Is the requirements specification of the solution formal?

   Since all tests are specified in a formal description technique, the requirements specification is formal.

3. Does the solution comprise a functional design?

   We have functionally designed only the control unit. Its specification can be used as a template for an implementation.

   All components of the physical system are described mainly in terms of their externally visible behavior.

4. Has the functional design of the solution been verified against the requirements specification?

   The LOTOS specification of the steam boiler system has been formally and automatically validated and verified by the use of LOTOS tools. The TE-LOTOS specification and the Gofer specification have been formally but manually verified.

5. Does the solution comprise an architectural design?

   The specification defines an architectural design for the control unit and for the pumps. They are decomposed internally into a set of concurrent and communicating processes.

6. Has the architectural design of the solution been verified against the requirements specification?

   The same answer as to question 4.

7. Does the solution comprise an implementation of a control program?

   No, but the control unit structure is very close to that of an implementation

8. Has the control program been linked to the FZI simulator?

   No.

9. Has some experimentation been done with the control program?

   No.

10. How much time has been spent on producing the solution? Give the number of person months, if possible for the various parts of the solution.

    We needed approximately two weeks for developing the specification of the physical system (especially for the correct behaviour of the pumps), and another three weeks for the control unit.

    We spent the most time for verifying our solution, since we have to develop a version with a finite state space and with reduced use of data. But even with reduced (finite) state space our tools need several days.

11. How much preparation is needed to become sufficiently expert of the used specification framework in order to produce a solution to such a problem in that framework? Indicate the number of weeks training which you believe is needed for an average programmer to learn the method.

    We think, that an average programmer can learn about the used formalisms in eight weeks. Another important point is that some familiarity with the problems of designing safe real time systems is needed. It can take several weeks for acquiring some experience in real time systems.

12. What are the premises for a good understanding of the proposed solution?

    - Is detailed knowledge of the used formalism needed?

      Without deeper knowledge in process algebraic formalisms (like CSP or CCS) it is not possible to understand our specification and the verification methodology. However, there are some similarities between important concepts in process algebras and some concepts used in imperative programming languages (e.g. action prefix vs. statement lists).

      We have used the functional language Gofer for the data dependent part of our specification. Except the concepts of type classes and parametric polymorphism, an average programmer should be able to have an intuitive understanding of the function definitions and the computed values, since the well known mathematical syntax is widely used in Gofer.

    - Can an average programmer understand the solution?

      Without knowledge of the used formalisms it is not possible to understand all the necessary details. However, we think its easy to understand the basic ideas of our solution easily.

- How much time do you believe is necessary for the average programmer without knowledge of the used specification method to learn what is needed to be able to understand the solution?

  The used formalism seems to be simple, because only a few operators are available, but without some practice in using this formalism it is not possible to reach the necessary deeper understanding. We think, that an average programmer will need one or two weeks for the TE-LOTOS part. After this time he should be able to read and to understand the formalism.

  The time needed for the data type part in Gofer depends on how many knowledge of functional programming concepts is available. A newbie in functional programming needs two or three weeks, to get familiar with the fundamental concepts of lazy programming languages under the premise of good knowledge in imperative languages.

6 Conclusions

TE-LOTOS is a powerful method for the specification of hybrid real-time systems. The steam boiler specification is very concise and well understandable. The modular specification is well supported by TE-LOTOS with several structuring mechanisms such as process hierarchies and the definition of action visibility. The disabling operator can elegantly be used to describe exception handling mechanisms.

TE-LOTOS allows us to specify a system on different abstraction levels and the use of different specification styles. In particular, we successfully applied stepwise refinement. Several methods for formal reasoning exist, from which we selected the testing method to verify the correctness of the steam boiler specification.

Besides the advantages of TE-LOTOS, we identified two main drawbacks of TE-LOTOS

- The algebraic data type language Act One is to clumsy, so that we replaced it with Gofer. In fact, actual investigations on the new generation of LOTOS consider the inclusion of functional data type languages (in particular a subset of EML).

- The lack of priorities forbids the description of instantaneous timeouts. We had to introduce a marginal delay `epsilon` in order to react on missing messages.

The lack of tools for TE-LOTOS makes it currently inapplicable to real problems. However, their development is under work.

Last but not least, we must admit that the learning curve both for developing and understanding a TE-LOTOS specification is relatively large. But this is acceptable when compared with the advantages of TE-LOTOS.

References

[ABL95] J.-R. Abrial, E. Boerger, and H. Langmaack. Methods for semantics and specification. Preliminary report for the Dagstuhl-Seminar 9523, 1995.

[Abr95] J.-R. Abrial. Additional information concerning the dynamic behaviour of the steam boiler, 1995.

[BvLV95] T. Bolognesi, J. v.d. Lagemaat, and C. Vissers. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publisher, 1995.

[Eer94] H. Eertink. *Simulation Techniques for the Validation of LOTOS Specifications*. PhD thesis, University of Twente, 1994.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1*. Springer Verlag, 1985.

[Gar95] H. Garavel. *CADP — The Caesar/Aldebaran Distribution Package*. CNRS/IMAG and INRIA, 1995.

[HJea92] P. Hudak, S.P. Jones, and P. Wadler et al. Report on the programming language haskell, a non-strict, purely functional language, version 1.2. Report, mar 1992.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Endlewood Cliffs, N.J., 1985.

[ISO88] ISO/IEC, Geneva, Switzerland. *IS 8807: Information Processing Systems – Open Systems Interconnection – LOTOS – A formal Description Technique based on the temporal Ordering of observational Behaviour*, 1988.

[Jon93] M.P. Jones. GOFER functional programming environment, version 2.28. Report, 1993.

[LL94] G. Leduc and L. Leonard. A formal definition of time in LOTOS. Draft version, aug 1994.

[LLdFQ95] G. Leduc, L. Leonard, D. de Frutos, and J. Quemada. Time extended LOTOS. Revised Working Draft on Extended LOTOS, ISO/IEC JTC1/SC21/WG7, jul 1995.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer Verlag, New York, 1980. LNCS 92.

[pE93] LotosPhere project (ESPRIT 2304). *LITE — The LOTOS Integrated Tool Environment*, 1993.

[QMFL94] J. Quemada, C. Miguel, D.d. Frutos, and L. Llana. A proposal for timed LOTOS. Revised Draft on Enhancements to LOTOS, J. Quemada (ed.), 1994. ISO/IEC JTC1/SC21/WG1.