Specifying the Remote Controlling of Valves in an Explosion Test Environment

Martin Schönhoff* and Mojgan Kowsari**

Technical University of Braunschweig, Computer Science, Databases,
Postfach 3329, D–38023 Braunschweig, Germany,
email: M.Kowsari@tu-bs.de

Abstract. We present parts of the specification of a program to remote control and monitor different devices, especially valves, in an explosion test environment. The program was developed within an industrial national project called CATC carried out in PTB, the German federal institute of weights and measures. The CATC information system supports various activities of different user groups that are responsible for testing and certifying explosion proof electrical equipment in PTB. Our approach is based on the formal object-oriented specification language Troll. We describe the advantages of the use of the formal method in our project.

1 Introduction

In the past few years, there has been considerable activity in the area of modelling large information systems. Many industrial methods have been developed for every platform and for different users, local or in networks. But they do not reach the level of formality achieved by formal specification languages. One main problem remains when designing a real world aspect: "Do we get what we need?" There is a small but growing community of people who propose and promote formal methods in software engineering [BH94]. The acceptance of formal methods in industry is still low. This is mainly due to the fact that formal methods are thought to be complex, hard to handle, and not suitable for real world applications [FBGL94, BH95].

In this paper, we present our experiences with the formal specification language Troll, gained while using Troll to design a large information system in an industrial environment [KHDE96, KKH$^+$96, Kow96]. Troll helps to discover and eliminate ambiguities and vaguenesses in the modelling phases. When we started our project in 1994, no formal method was applied. Soon some problems arose [HS94], and mid 1995, we became aware that the project was already

* Now at University of Zurich, Department of Computer Science, Database Technology Research Group, Winterthurerstr. 190, CH–8057 Zurich, Switzerland, email: mschoen@ifi.unizh.ch

** Work reported in this paper is supported by the Physical Technical Federal Board, lab 3.51, project Computer Aided Testing and Certifying

likely to fail. One of the problems of informal methods we encountered was that they require the designer to think about implementation aspects. However, our application domain and its data are too complex to mix design and implementation without loosing the global view of the system. Hence, we decided to use a formal approach. Using formalism allows us to concentrate more on the data and data structure and to determine what the system has to do under exceptional circumstances. Due to safety-critical aspects of our problem domain, emphasis on this point was especially important and useful in the process of requirement acquisition of the remote controlling of the valves.

The TROLL approach incorporates many ideas which have been developed over the past eight years. TROLL supports the declarative specification of conceptual models. TROLL defines an abstract model called the *Universe of Discourse* to cover all aspects which are relevant with respect to organisational activities in complex information systems. It includes the functional requirements of the later system and excludes non-functional requirements (like technology bindings of later implementations).

The remainder of this paper is structured as follows. Section 2 provides a summary of the concepts of TROLL. In Sect. 3, we give a short introduction to the problem domain of testing electrical apparatus in flameproof enclosures. Some of the requirements for VENTIL are presented in Sect. 4, while Sect. 5 shows the resulting TROLL specification. Our experiences are discussed in Sect. 6. Finally, Sect. 7 concludes the paper.

2 TROLL

In this section, we give a short introduction to the specification language TROLL.

TROLL ("Textual Representations of an Object Logical Language") is a formal language for the specification of object systems on a high level of abstraction. The basic ideas and concepts of TROLL can be summarised as follows:

- The basic building blocks of information systems are objects.
- Objects are classified into classes and described by a set of attributes and actions.
- Every object describes a set of *sequential life cycles*, i.e. sequences of *local actions* on the object.
- An object system is composed of a number of concurrent objects. These objects are the *nodes* of the system. Nodes usually have other objects as components. To establish global communication in an object system, nodes can be connected through global interactions.

The following are the basic features of the language:

- A *system specification* consists of a set of *data type* definitions, a set of *object class* specifications (prototypical object descriptions), and a number of *object declarations*.

- *Parameterised data types* allow for the construction of new data types based on a fixed universe of predefined data types.
- An object class specification is a set of *attributes*, *actions*, and *constraints*.
- Object classes may be constructed over other object classes (*aggregation*) to describe complex objects, i.e. objects which contain component objects.
- An object class may be the *specialisation* of another object class. The specialised class (*subclass*) may have properties in addition to those inherited from its *superclass*.
- Concurrent objects are declared over object classes. These declarations describe the potential objects in the system. Interactions (through *action calls*) between different objects describe the global synchronisation relations. All actions which are called within one *event* are understood to take place concurrently. Action parameters are exchanged through unification.

The case study which will be introduced in Sect. 4 illustrates some of the language features. For more details, see [Har97, DH97].

Semantics are assigned to TROLL specifications using different techniques: The static structure of an object system is semantically described with algebraic methods, statements over object states are expressed with a logic calculus, and the dynamic structure of the system, i.e. its evolution, is reflected via a temporal logic which is interpreted in terms of event structures. For an exhaustive description of the underlying theory, semantics, and logics see [Ehr96, ES95, EH96], for the refinement of object specifications refer to [Den96, Den95].

3 Problem Domain

In this section, we provide a short introduction to the application domain. We offer basic information about electrical apparatus in flameproof enclosures and the explosion test environment needed to certify them.

The Physical Technical Federal Board (PTB) [RBH87] is a federal institute for science and technology and the highest technical authority for metrology and physical safety in Germany. Its tasks are research in physics and technology, realisation and dissemination of SI units[3], cooperation in national and international technical committees, physical safety engineering serving against explosions, etc.

The PTB's group 3.5 "explosion protected electrical equipment" is concerned with the testing and certifying of explosion proof electrical equipment. Such equipment may only be used in hazardous areas after it has been approved and certified following the harmonised European standards EN 50014–50028. The assessment procedure consists of testing the formal and informal documents, checking the design papers (technical drawings) and experimental tests (such as explosion, flame propagation, and thermal-electrical tests). Currently, all steps which are necessary during the testing procedure and the issuing of about 1000 certificates each year are carried out manually and individually by the approximately 100 employees who make up the three labs of group 3.5. Because of

[3] international system of units

the huge amount of data, a standardised archive and catalogue of all existing certificates of explosion proof equipment is planned. It will be integrated into a software package called CATC (Computer Aided Testing and Certifying). Since 1994, the design and modelling of CATC is the long-term aim of the cooperation between the PTB and the database group of the Technical University of Braunschweig.

CATC has to support three different problem domains: *administration management*, *design approval*, and *experimental tests*, which are performed in a test lab. CATC is not a standalone information system, but it has to be embedded into an existing environment. Besides, we have to deal with existing application programs which have to be re-specified because they are erroneous. These re-specified parts have to be embedded into the new information system structure. In addition, there is a link to the frequently accessed PTB-wide database. To summarise, we have a safety-critical application area that comprises both technical and database aspects in a complex heterogeneous environment as well as existing and re-developed applications.

4 Requirements for VENTIL

This section focuses on some requirements for VENTIL[4], a program to remote control and monitor explosion test stands in the test lab of group 3.51. It is a part of the experimental test software of CATC. PTB's group 3.51 deals with the certification of *electrical apparatus in flameproof enclosures* according to the standards EN 50014 and EN 50018 [EN 87a, EN 87b] (motors, pumps, and switches, for instance). For *flameproof enclosure*, all parts which can ignite an explosive atmosphere are placed in an enclosure. In the case of an explosion inside, the enclosure withstands the pressure developed and prevents the transition of the explosion to the surrounding explosive atmosphere. The critical places for explosion transition are the *joints*, the places where corresponding surfaces of two parts of an enclosure come together and therefore a *gap* arises. For a flameproof enclosure, each gap must be narrow enough so that only *flameproof joints* are formed [ORW83].

Consequently, *flameproof joint tests* are among the experiments undertaken in the test lab to certify flameproof enclosures [EN 87b]. In a flameproof joint test (Fig. 1), a prototype (2) is placed inside the test chamber (1, called an *autoclave*) of an *explosion test stand* and is filled with an explosive atmosphere (eA). Then, a spark (S) ignites the atmosphere inside the enclosure. A prototype passes the test if the enclosure withstands the developing pressure and temperature (°C) and the explosion does not continue into the autoclave.

The main equipment of an explosion test stand are a gas source, the autoclave, analysis tools, pumps, and valves. All of these devices are connected in a

[4] This is German for "valve". The name originates from the previously used program which only let the user open and close valves. The name VENTIL was kept for the newly developed application because of habit.

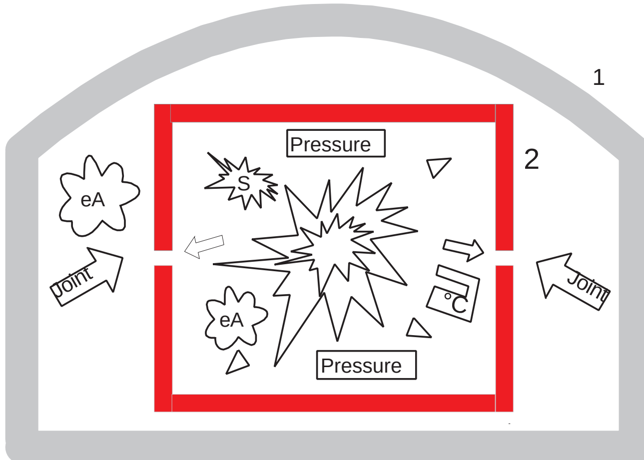

Fig. 1. Flameproof joint test.

net of tubes and pipes. Figure 2 shows a schematic view of the smallest explosion test stand in the test lab, the so called *Ex–Eva*[5]. VENTIL is used to control and monitor most of the devices of the Ex–Eva in order to create explosive atmospheres in the autoclave. The actual measuring of explosion pressure during an experiment is done separately [Hoh96, Sch96a].

Besides the obvious tasks to let users (i.e. the *testers*) mix gases, open, close, and monitor valves, turn on and off pumps, etc., VENTIL provides two more advanced features: the automatic *observance of dependencies* between devices and the *calculation of the gasflow*.

4.1 Observance of Dependencies

VENTIL prevents testers from accidently violating *dependencies* between devices. Dependencies are rules which have to be observed to protect the equipment and the environment (including the testers themselves) of the explosion test stand. The dependencies can be formulated as a kind of "master-slave" functions — one device depends on the state or state change of one or more other device(s).

These example dependencies (to which we will refer throughout Subsect. 5.2) are needed to protect the fragile oxygen analyser of the Ex–Eva (cf. Fig. 2) from extreme pressure and soot developed in the autoclave during an explosion:

1. Valve 31 may be open if and only if valve 26 is open.
2. (a) Before valve 31 is opened, valve 34 is opened automatically.
   (b) Before valve 34 is opened, valve 35 is opened automatically.
   (c) One second after valve 31 has been closed, valve 34 is closed automatically.
   (d) After valve 34 has been closed, valve 35 is closed automatically.

[5] **E**xplosions–**V**ersuchs **A**nlage (German for "explosion test stand")

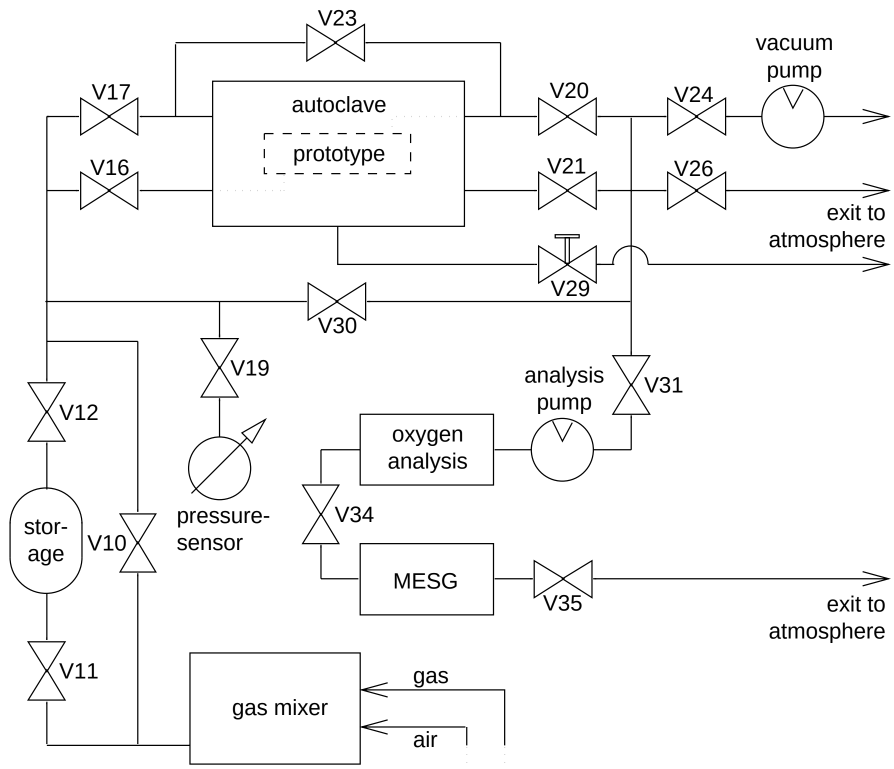

Fig. 2. Schematic view of the Ex–Eva. Vxx denotes a valve.

4.2 Calculation of the Gasflow

In a schematic display similar to the one of the Ex–Eva in Fig. 2, VENTIL offers to the testers a calculated view of the gasflow in the test stand. Unfortunately, this calculation is not at all trivial, but depends on parameters like the expected (yet not measured) gas pressure and the pumping direction of pumps. We will not go into the details of the parameters, but we use significantly simplified requirements for the visualisation of the gasflow here:

- A gasflow may begin or end at any gas entry or exit point of the test stand (e.g., the exit to the atmosphere) as well as at the gas mixer, reservoir, autoclave, or pressure sensor (altogether referred to as *endpoints*).
- There has to be an "open way" from one endpoint to another, i.e. all valves need to be open and the pumps turned on in a gasflow. All the other devices, the oxygen analyser, for instance, do not influence the gasflow and are treated here simply like a pipe.

Nevertheless, these reduced requirements will still be sufficient to present the implications of the gasflow calculation as far as this paper is concerned. An unabridged description is given in [Sch96b].

5 Specification of VENTIL using Troll

The main components of the Troll specification of VENTIL are presented in this section. First, a general overview of the object hierarchy of the information system node is given. Afterwards, the two most interesting parts of the specification are treated in detail: the observance of dependencies and the calculation of the gasflow. The diagrams illustrating this section use a notation similar to OMT [RBP+91] which was adapted to Troll [JWH+94, WJH+93].

5.1 Overview

The specification of the VENTIL system is made up of three *nodes* (cf. Sect. 2), namely the user, hardware, and information system nodes. The user node describes the possible behaviour of the different user groups (testers, technicians, etc.) and their interfaces to the main system. For instance, in the `Tester`[6] object class (which is a part of the user node) it is specified that valves can be opened and closed or what data must be provided for the gas mixer. These specifications solely focus on functionality and data and are therefore abstractions of possible implementations (like dialog boxes or other user interface elements). Digital outputs (e.g., "open valve"), digital and analogue sensors ("valve is open", voltage representing measured pressure), etc. are modelled in the hardware node.

One merit of specifying VENTIL in Troll is the possibility to examine the information system node isolated from the nodes describing user interaction [Sch96b] and hardware behaviour [Hoh96]. In this paper, the latter nodes and global interactions are not treated any further. We only discuss the specification of the information system node, beginning with the introduction of its object classes in the remainder of this subsection. The Community Diagram in Fig. 3 gives an overview of the component and inheritance hierarchies used.

The Object Class `Knot` The calculation of the gasflow requires the most complex algorithm in VENTIL. Hence, the structure of the specification has been designed to suit this algorithm best. From Fig. 2 and the description in Subsect. 4.2 it is rather obvious that a gasflow can be formalised as a path in a *directed graph* representing the explosion test stand. The *nodes* of the graph stand for the devices[7] of the test stand, and the *vertices* for its pipes[8]. Although pipes are generally undirected, the graph's vertices need to be directed here, because at one time gas can only flow one way, determined by the pumping directions of the pumps and the gas mixer.

All basic properties of a node in the graph are modelled in the abstract (i.e. not instantiable) object class `Knot`. It is a superclass of any object class

[6] Throughout this paper, we print all terms referring to the Troll specification in `typewriter` font and Troll keywords in *italics*.

[7] Subsequently, devices also subsume the joins between two or more pipes, and the entry and exit points of the test stand (e.g., the external gas supply).

[8] To simplify reading, we will no longer distinguish tubes from pipes.

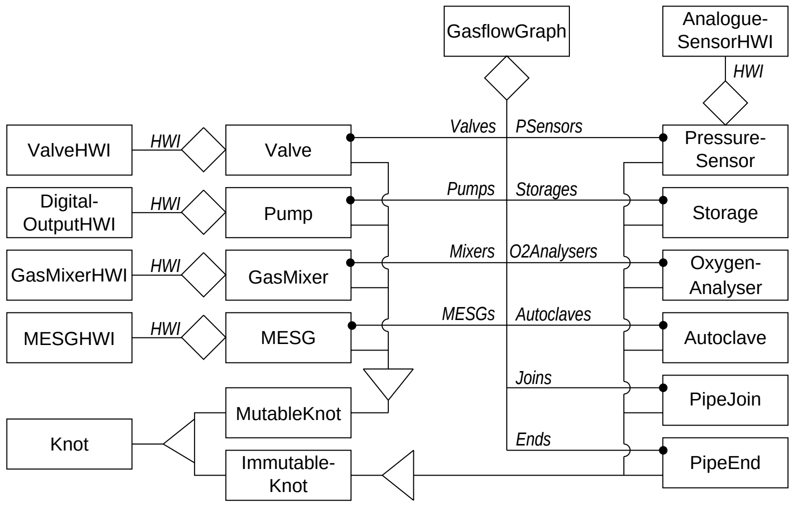

Fig. 3. Community Diagram of the information system node of VENTIL. The triangles symbolise inheritance, the diamonds component relationships. The dots are read as "zero or more" components.

representing a concrete device[9]. Hence, the devices become nodes of the graph, but do not need to take care of their connections to other nodes or their behaviour during the gasflow calculations themselves. This is an excellent example for the use of inheritance in TROLL: Each device class inherits the basic properties of a `Knot`. Evolution within these properties does not have any effect on object classes apart from `Knot`, thus facilitating the maintenance of the specification a great deal. All devices (including those which may be added to the test stand in future) reuse the specification of `Knot` and are therefore modelled more quickly and understandably. Furthermore, a `Knot` does not need to know which kinds of devices it is connected to, because it does not need any specialised properties of its neighbouring `Knots`.

Here is a part of the TROLL specification of `Knot`:

```
data type vertex = record(knot:|Knot|, flow:bool)
data type names = string(3)
data type switch = enum(activate, deactivate)
object class Knot
attributes Vertices: set(vertex) isConstant;
           Type: enum(endpoint, through) isConstant;
           Status: enum(closed, opening, open, closing);
           Name: names;
```

[9] While we are discussing the specification of VENTIL, we will use the name of a real-world object synonymous to its representing TROLL object; e.g., by "valve 31", we generally mean "the object representing valve 31 in the specification". The few exceptions are made clear through phrases like "the hardware of valve 31".

```
actions FindFlow(visited:set(|Knot|), flow:set(|Knot|),
             ! newFlow:set(|Knot|), ! success:bool);
        FindFlowNo(no:nat, vertices:list(vertex), visited:set(|Knot|),
               flow:set(|Knot|), ! newFlow:set(|Knot|), ! success:bool);
        Switch(action:switch, ...)  -- for the second parameter, see Subsect. 5.2
constraints cnt(Vertices) > 0,
          cnt(Vertices) = cnt(toSet(select v.knot from v in Vertices)),
          all vert in Vertices (vert.knot # self);
end;
```

We do not use a vertex object class in the specification of VENTIL. It is sufficient to keep a set of references to neighbouring `Knots` (together with a flag denoting whether this `vertex` is in the gasflow or not) to store outgoing vertices[10]. The three constraints on `Vertices` make sure for each `Knot` (i) that it is connected to at least another one, (ii) that there are no two vertices to the same `Knot`, and (iii) that there is no vertex to itself ((i) to (iii) are always fulfilled for an explosion test stand). Subclasses of `Knot` add constraints according to their specialised needs: A pump, for example, must always have one incoming and one outgoing vertex to denote the pumping direction. Note how simple allowed states of an object can be defined in TROLL. Constraint (ii) also serves as one of many examples in VENTIL where the power of the descriptive *select* statement is exploited to yield a compact specification.

The constant attribute `Type` specifies whether a `Knot` is an `endpoint` of a gasflow or the flow just runs `through` the `Knot`. The `Status` attribute stands for the states different devices may take. For a valve, `open` means it is open, `opening` that it is no longer closed, but not yet open (due to the mechanical switching delay) and so on. Figure 4a shows the Object Behaviour Diagram of `Valve`; due to mechanical malfunction, any state transition is possible. The `Status` of a pump can only be either `open` (turned on) or `closed` (turned off) (Fig. 4b) — enforced by a constraint. The `Type` and `Status` attributes and the actions `FindFlowNo` and `FindFlow` are needed in the gasflow calculation and are treated in detail in Subsect. 5.3.

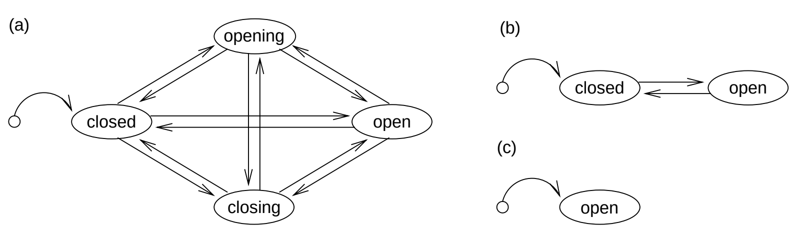

Fig. 4. Object Behaviour Diagrams: (a) `Valve`, (b) `Pump`, (c) `ImmutableKnot`.

[10] Specifying vertices is more complicated with the complete gas flow algorithm (using gas pressure, cf. Subsect. 4.2), because the graph needs to be traversed along incoming vertices as well. A direction part is added to the `vertex` data type and an additional constraint is needed to control the resulting redundancy [Sch96b].

The enumeration `switch` generalises the notions of "opening a valve", "turning on a pump", etc. to `activate` and the respective counterparts to `deactivate`. It is used as the first parameter to the action `Switch` which is overloaded in any subclass of `Knot` to perform the required task for the individual subclass.

Finally, the `Name` is a user-defined identification of a `Knot`. It is simply a three character string like 'V11' for valve 11. For every operation a user likes to perform on a specific device, he inputs the `Name` to denote the device.

The Object Classes `MutableKnot` and `ImmutableKnot` Devices like the oxygen analyser or pressure sensors cannot be manipulated through VENTIL. With regard to their `Status` in the gasflow, those devices are always `open`. They are modelled as subclasses of the abstract object class `ImmutableKnot` which is a subclass of `Knot` (see Fig. 3). `ImmutableKnot` constrains the `Status` to `open` (Fig. 4c) and disables the inherited switching operation.

Devices that can be controlled by testers (e.g., valves, pumps) also have a common abstract superclass, `MutableKnot`. Obviously, only `MutableKnots` may need to observe dependencies, since only if the state of a device is mutable, it may depend on the state of another device. Hence, the observance of dependencies is handled in `MutableKnot`. See Subsect. 5.2.

The Device Object Classes The different device classes of the test stand are modelled as separate object classes in VENTIL. Each of these object classes is a subclass of either `MutableKnot` or `ImmutableKnot` and hence indirectly a subclass of `Knot`.

Several device object classes have components specifying hardware interfaces. By convention, the names of hardware interface classes all end on `HWI`. For instance, `ValveHWI` models the interface to an object class within the hardware node of VENTIL. `ValveHWI` provides actions to open and close a valve, to check the current status of the hardware, etc. A detailed introduction to the device and hardware interface object classes of VENTIL is beyond the scope of this paper. Refer to Fig. 3 for an overview and to [Sch96b] for details. However, it should be mentioned that TROLL served well in the description of `HWI`-classes which not only form the interface to the real hardware, but also to the work of another member of the CATC team [Hoh96].

The Object Class `GasflowGraph` The management of our graph and the initiation of the gasflow calculation is modelled in the object class `GasflowGraph`. Here follows the part of its specification relevant for this paper:

```
object class GasflowGraph
components Valves: map (names) to (|Valve|);
           Pumps: map (names) to (|Pump|);
           ...
attributes Knots: map (names) to (|Knot|) derived
           Knots(name):= select knot from knot in dom(Valve)+dom(Pump)+...
                         where knot.Name = name;
```

```
actions Gasflow();
        GasflowNo(no:nat, knots:list(|Knot|), flow:set(|Knot|),
                    !newFlow:set(|Knot|));
constraints all name in names (
               cnt(select knot from knot in dom(Valves)+dom(Pumps)+...
                   where knot.Name = name) <= 1);
end;
```

In the *components* section, parametrised components are declared for each of the device subclasses of `Knot`. The parameter domains are always the range of possible `names` for `Knots`. In TROLL, it is necessary to specify the exact class of a component and not just one of its superclasses. It is therefore not possible to have one parametrised component containing instances of any of the subclasses of `Knot`. But since all `names` within a test stand are supposed to be unique even for different device classes, a well-defined map from `names` to `Knots` is required. It is achieved through the constraint given above which states that each `name` may appear at most once in the union of all domains (i.e. the actually existing instances) of the parameterised components. Convenient access to the map from `names` to `Knots` is provided through the derived attribute `Knots`. The two actions `Gasflow` and `GasflowNo` initiate the search for gasflows in the graph. They are treated in detail in Subsect. 5.3.

5.2 Observance of Dependencies

Classification of Dependencies The formalisation of dependencies (like those of the examples in Subsect. 4.1) leads to the distinction of three types: static, dynamic, and delayed dependencies[11].

Static dependencies involve at most *one state change* in one device. This state change depends on the state of another device which is only watched, but not changed. Example 1 is a static dependency: Valve 31 may only be open if and only if valve 26 is open.

Dynamic dependencies always involve the possibility of *two state changes* in two devices, *as fast as possible*. From the point of view of one of the involved devices, there are three possible executions of the own state change: *before* or *after* the other device or both in *parallel*. Specifying the parallel and after cases is straightforward. For the before case, we take a look at Example 2b, where valve 35 must be opened *before* valve 34. The two following TROLL events must take place if valve 34 is commanded to open itself:

1. If valve 35 is already open, valve 34 opens and nothings else needs to be done. Otherwise, valve 34 commands valve 35 to open.
2. As soon as the hardware of valve 35 is opened, its corresponding object is notified and commands valve 34 to open.

[11] Following the vocabulary of the engineers in lab 3.51, there is also a fourth type of "dependencies" in the original requirements analysis. But its formal definition revealed that it must be treated differently from the other three [Sch96b].

Delayed dependencies are a special case of dynamic dependencies. They also involve the possibility of *two state changes* in two devices, but introduce a *delay time* between the switching operations. Obviously, *parallel* delayed dependencies do not make sense, thus leaving the *before* and *after* cases.

Delayed dependencies are treated similarly to the other dynamic dependencies, but another event is added. In Example 2c, valve 34 has to be closed one second after valve 31 is closed. Listing the required TROLL events for the closing command on valve 31, we get:

1. Valve 31 closes.
2. As soon as the hardware of valve 31 is closed, its corresponding object is notified. If valve 34 is already closed, nothing else needs to be done. Otherwise, the delay time begins.
3. As soon as the delay time has expired, valve 31 commands valve 34 to close.

Modelling Dependencies with Duties The observance of any type of dependency is modelled in a system of *duties*. One dependency can result in a number of duties imposed on several devices (e.g., see below how Example 1 is treated). Duties are specified as *record*-types in VENTIL. They are stored as attributes in the *duty list*[12] of the `MutableKnots` they are imposed on. The duty list is checked before any switching operation is applied to the device. A duty *object class* would not be helpful, because all actions which process duties only modify attributes of `Knot`, but never the values of a duty (except for creation and deletion, of course).

Duty types are modelled as follows in TROLL:

```
data type execution = enum(now, before, parallel, after)
data type duty = record(trigger : switch,    exec : execution,
                        delay : time,        target : |Knot|,
                        action : switch,     once : bool)
data type delayedDuty = record(time : time,
                               duty : duty)
```

The enumeration `execution` is used to distinguish static (`now`) from dynamic `dutys`. In the latter case, the time of execution of the second state change is given as either `before`, `after`, or in `parallel` with the first state change, as explained above.

The first component of the `duty` *record* holds the information on which `switch` the `duty` must be fulfilled; e.g., a `duty` with the `trigger` value `activate` imposed on a valve must be fulfilled each time the valve is opened. The `exec` component determines the type of the `duty`. For `before` and `after` duties, `delay` holds the time between the first and second switching operation; a delayed dependency has a value greater then 0. To fulfill the `duty`, `action` has to be passed to the `Switch` operation of the `target`. The flag `once` is set for `dutys` that have to be removed from the duty list as soon as they are fulfilled. A `delayedDuty` is an ordinary `duty` which has to be fulfilled at a certain system `time`.

[12] The name "duty *list*" emerged during development although no sequencing is needed; see the declarations for `MutableKnot` below.

Fulfilling Duties The declarations of `MutableKnot`, as far as the observance of dependencies is concerned, look like this:

object class **MutableKnot**
aspect of `Knot` *on* ... -- `Knot` is the superclass of `MutableKnot`
attributes `DutyList : `*set*`(duty);`
`DelayedDutyList : `*set*`(delayedDuty);`
...
actions `AreDutiesFulfilled(trigger:switch, `*!*` now:`*bool*`, `*!*` before:`*bool*`)`
`FulfillDuty(duty:duty)`
`FulfillAllDuties(trigger:action, exec:execution)`
`FulfillDelayedDuties()`
`Switch(action:switch, duties:`*set*`(duty))`
...
end;

`AreDutiesFulfilled` returns (denoted by a '*!*') for a given action whether all `now` and `before` `duty`s in the `DutyList` are fulfilled. The return values are used by the switching operation of specialised `ImmutableKnot`s to determine whether the desired `action` is allowed now or later or must be rejected. `FulfillAllDuties` calls `FulfillDuty` to fulfill all `duty`s in the `DutyList` for the given `trigger` and `exec` parameters, e.g., to fulfill all `duty`s `before` the `MutableKnot` is `activated`. Similar to `FulfillAllDuties`, `FulfillDelayedDuties` is used to process the `delayedDuty`s in the `DelayedDutyList` as soon as their delay time has expired.

The action `Switch` is inherited from the superclass `Knot` (see Subsect. 5.1). Here, we introduce the second parameter, the set `duties`. All members of `duties` are added to the `DutyList`. Usually, `duties` is empty, but to fulfill a `before` `duty`, one new `duty` is passed; see below.

Lead by the examples introduced earlier, we will now take a look at how these actions work together if a static, dynamic, or delayed dependency must be fulfilled.

Fulfilling a static dependency is as simple as expected. Example 1 requires two `duty`s:

`(activate, now, 0, `*Valve 26*`, activate, `*false*`)`

imposed on `Valve` 31 and

`(deactivate, now, 0, `*Valve 31*`, deactivate, `*false*`)`

imposed on `Valve` 26.

While opening, the first `duty` must be fulfilled for `Valve` 31. The `Switch` action of `Valve` 31 checks the `Status` of the `duty`s `target`, *Valve 26*[13]. If `Valve` 26 is `activated` (i.e. the `Status` of `Valve` 26 is not `closed`), the hardware of valve 31 can be `activated`, too. Otherwise, the switching command is rejected. Similarly, `Valve` 26 must check the status of `Valve` 31 before closing (according to the second `duty` given above).

The dynamic dependency of Example 2b results in the `duty`

[13] This is meant to be the identity of the object representing valve 26.

(activate, before, 0, *Valve 35*, activate, *false*)

imposed on Valve 34.

What happens if the dynamic dependency of Example 2b must be fulfilled is shown in the Object Communication Diagram in Fig. 5. In the first TROLL event (shown as continuous arrows), the Switch action is called to open Valve 34. Switch uses AreDutiesFulfilled to find out that there is at least one unfulfilled before duty and calls FulfillAllDuties with the parameters activate and before. FulfillAllDuties calls FulfillDuty to fulfill all necessary dutys, including the one of our example above. To fulfill the duty, Switch is called for Valve 35. The arguments passed are activate to open the valve (what is done by a call to the hardware interface object) and a *set*(duty) containing

(activate, after, 0, *Valve 34*, activate, *true*).

This new duty is added to the DutyList of Valve 35.

As soon as the hardware of Valve 35 is opened, the second TROLL event is initiated (dashed arrows). Because the event takes place after Valve 35 has been activated, the new duty must be fulfilled and then deleted from the DutyList (the last component of the duty is *true*). Fulfilling the duty results in the opening of Valve 34, thus we have Valve 34 opened after Valve 35 — as required by the dependency.

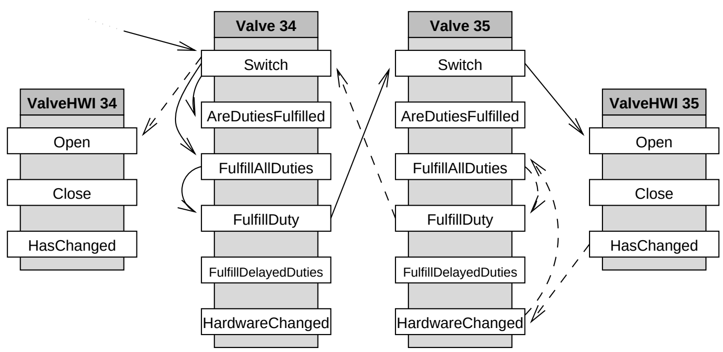

Fig. 5. Object Communication Diagram for Example 2b (dynamic dependency).

Finally, we take a glance at Example 2c and the respective Object Communication Diagram (Fig. 6). The duty which is imposed on Valve 31 is

(deactivate, after, 1, *Valve 34*, deactivate, *false*).

If the duty must be fulfilled (event one, continuous arrows), the hardware of Valve 31 can be Closed immediately, because we talk about an after duty. After Valve 31 is closed, it tries to FulfillAllDuties (event two, dashed arrows). It therefore delays our example duty by adding

(*current time + 1 sec*, (deactivate, after, 1, *Valve 34*, deactivate, *false*))

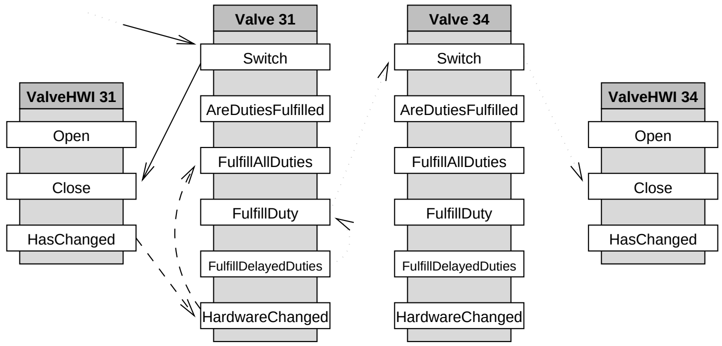

Fig. 6. Object Communication Diagram for Example 2c (delayed dependency).

to the `DelayedDutyList` of `Valve` 31. As soon as the specified time is reached the third event takes place (dotted arrows): `FulfillDelayedDuties` calls once again `FulfillDuty`, and `Valve` 34 is `Closed`, too.

5.3 Calculation of the Gasflow

Searching the Gasflow Based on the requirements mentioned in Subsect. 4.2, we derived the algorithm to calculate the gasflow in an explosion test stand. In short, an action `Gasflow` searches all `Knots` in the `GasflowGraph` and determines whether they are endpoints or not. If a `Knot` is an endpoint, `Gasflow` finds all directed, acyclic paths (called *gasflows*) to a different endpoint, leading only through `Knots` with a `Status` different from `closed` (all `ImmutableKnots`, for instance).

This search is done by calling the recursive action `FindFlow` in the endpoint. In the case of a successful search, `FindFlow` returns the set of `Knots` which are contained in any of the gasflows beginning in this endpoint. The union of all those sets for all endpoints obviously contains all `Knots` through which gas flows. This union is called *the* gasflow, because the following can be shown: The vertices between any of the `Knots` in the gasflow represent exactly those pipes of the test stand which contain gas. This means, to show the gasflow in the test stand to the users, it is sufficient to mark the vertices between each two `Knots` in the gasflow.

Recursive Search for Endpoints We stated above that `Gasflow` must find each endpoint of the graph, call `FindFlow` for them, and calculate the union of the resulting gasflow sets. To start with, it is necessary to explain the signature of `FindFlow`:

```
FindFlow(visited:set(|Knot|), flow:set(|Knot|),
         !newFlow:set(|Knot|), !success:bool);
```

The set `visited` contains all `Knots` of the graph which are already part of the current recursion. This parameter is used to avoid cycles in the search. The other input parameter, `flow`, holds the set of `Knots` which have already been discovered to be in the gasflow currently searched. Corresponding to `flow` is the output parameter `newFlow`. In this set, all the members of `flow` are returned plus the identity of the current `Knot`, if it is in the gasflow, too. In this case, `success` is set to *true*.

For the specification of `Gasflow` it is therefore necessary to call

```
inGasflow : set(|Knot|);
ignore : bool;
FindFlow({}, {}, inGasflow, ignore);
```

to each endpoint of the graph and unite the resulting `inGasflow` sets. The `success` parameter can be ignored here.

The calls to `FindFlow` and the collection of their results is easily specified if we can use command sequences. But with TROLL, we encounter the challenge that the whole calculation has to be done in *one shot*. There is no *while* statement. There is no way to make an arbitrary number of action calls and keep the results for "later" processing. The only possibility we have is to use the unification of action calls to imitate a sequence of calls. In such a "sequence", the `newFlow` result of a call has to be inserted as the `flow` parameter of the next call.

We use the additional recursive action

```
GasflowNo(no:nat,knots:list(|Knot|),flow:set(|Knot|),! newFlow:set(|Knot|))
```

for this task. The first two parameters control the recursion: `knots` contains a *list* of all `Knots` in the graph (in arbitrary, but fixed order), `no` the current index in the list. The recursion is initiated with the parameter `no` set to the number of `Knots`. The index is decremented in each recursion step until 1 is reached. The parameters `flow` and `newFlow` are used like their counterparts in `FindFlow`.

Here is the complete specification of `GasflowNo` and `Gasflow`. The *first* `Knot` for which `FindFlow` may be called is `knots[1]`. For the calculation of `newFlow`, note that the results `flowNo` and `flowOut` are undefined if the respective action calls do not take place.

```
GasflowNo(no:nat,knots:list(|Knot|),flow:set(|Knot|),! newFlow:set(|Knot|))
    variables flowNo, flowOut : set(|Knot|);
              ignore : bool;
    do onlyIf(no > 1): GasflowNo(no-1, knots, flow, flowNo);
       onlyIf(knot[no].Type = endpoint):           -- start recursing Knots here
         knot[no].FindFlow({}, no > 1 ? flowNo : flow, flowOut, ignore);
       newFlow := (knot[no].Type = endpoint) ? flowOut
                                             : (no > 1 : flowNo : flow);
    od
Gasflow()
    variables knots : list(|Knot|) derived knots := toList(dom(Knots));
              flow : set(|Knot|);
    do GasflowNo(length(knots), knots, {}, flow);          -- initiate recursion
       ShowGasflow(knots);                         -- visualise gasflow to users
    od
```

Recursive Search in the Graph and in the `Knots` We just discussed how the actions `Gasflow` and `GasflowNo` are used, so we do not need to cover any details of the `FindFlow` and `FindFlowNo` actions; they are used to recursively search the arbitrary number of `Vertices` within *a single* `Knot` analogous to the `Knots` of the graph. Additionally, `FindFlowNo` calls `FindFlow` in *a neighbouring* `Knot` to traverse the graph recursively.

Thus we have three nested recursions to calculate the gasflow — all of them carried out concurrently, only "sequentialised" through TROLL's unification mechanism. Figure 7 visualises the three recursions on instance level. Note that each `Knot` can be a part of several calling "sequences".

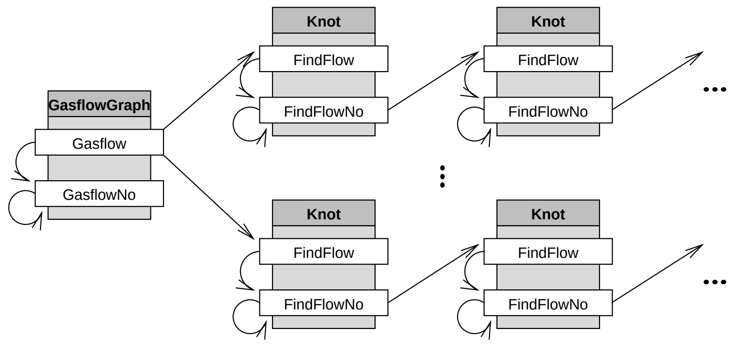

Fig. 7. Object Communication Diagram for the three nested recursions of the gasflow calculation.

6 Experiences

We have developed a specification of VENTIL that successfully exploits the features TROLL provides.

The object-orientation of TROLL helped us to find a modular structure and well-defined module interfaces. Inheritance became an important factor in the specification of the graph, since we were able to separate general properties of a node (e.g., during the gasflow calculation) from those of specialised device nodes. The general design and specification of a `Knot` is reused in every (future) device object class, and no change to some device class will influence other `Knots`.

Constraints, often in conjunction with the powerful descriptive *select* statement, proved to be very helpful, too. They allow for compact and yet simple restrictions of the possible behaviour of objects.

Because of the similarities to the transaction concept of databases, the virtues of parallelism are obvious if a "usual" information system is designed. But even in VENTIL, concurrent execution has advantages, e.g., whenever one device is the target of several duties. During the implementation, we were troubled by serialisation, because two duties that are fulfilled simultaneously in one TROLL event can have two different effects if they are fulfilled one after the other.

On the other hand, sequential execution would have been useful during the gasflow calculation. The implementation which we successfully derived from our specification has been simplified. It requires only one level of recursion instead of three in TROLL and is therefore probably easier to understand. We nevertheless specified the complete algorithm to allow for the *animation* of the whole VENTIL model[14]. Discovering that TROLL has deficiencies in the domain of VENTIL was not surprising though. Recall from Sect. 1 and 2 that TROLL is primarily designed for the development of information systems. As a rather technical application, VENTIL is situated at least on the edge of TROLL's target domain, if not even outside.

Finally, some more notes on the implementation are appropriate. From about 2500 lines of TROLL, we received an output of more than 20000 lines of C++ code. We derived rules describing how to translate many parts of the specification into C++ [Sch96b]. Although no tools where available, the transition from the compact TROLL notation to C++ was, except for the difficulties mentioned above, surprisingly straightforward. In fact — from the overall class structure to algorithmic details in the gasflow calculations — there are on any design level almost one-to-one relationships between the specification and the implementation. On the code level though, the direct translation of the specification required to implement many additional classes to support TROLL data types (like sets of `Knots`, etc.). Overall, we strongly believe that the formal specification of VENTIL has payed off.

7 Conclusions and Future Work

In this paper, we presented the specification of VENTIL, a program to monitor and control the devices in an explosion test environment. VENTIL is a part of the ongoing development of a large information system in the PTB. We also presented our experiences with the use of the formal specification language TROLL in our project. So far, they were positive. One advantage of using TROLL was to achieve first a rather global view before considering details. Changes to the finer grained specification documents did not affect the global view. Furthermore, the formality and clearly defined semantics of TROLL specifications carried over to the implementation. The work for one laboratory is finished and we are currently implementing systems for the other two laboratories [HDK+97].

In the next step of the development of TROLL, we will establish tool support [Gra97]. Most important are tools that allow for a fast modification of the specification documents while ensuring consistency throughout the project. The reification from specification to implementation is another objective we want to reach in the near future.

[14] At least theoretically, since there is no animation tool available for the current version of TROLL.

Acknowledgements

We thank the employees of the PTB's group 3.51 for their support, especially Achim Kühne for his patience in discussing the technical details of VENTIL. We also thank Peter Hartel and Grit Denker for answering specific questions on TROLL. Finally, we thank the anonymous reviewers for their useful hints on preparing the final version of this paper.

References

[BH94] J. P. Bowen and M. G. Hinchey. Seven more myths of fomal methods: Dispelling industrial prejudices. In M. Naftalin, T. Denvir, and M. Bertrani, editors, *FME'94: Industrial Benefit of Formal Methods*, number 873 in LNCS, pages 105–117. Springer-Verlag, Berlin, 1994.

[BH95] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(3):34–41, 1995.

[Den95] G. Denker. Transactions in object-oriented specifications. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types, Joint with the 5th COMPASS Workshop; S. Margherita, Italy*, number 906 in LNCS, pages 203–218. Springer-Verlag, Berlin, May 1995.

[Den96] G. Denker. Semantic refinement of concurrent object systems based on serializability. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 105–126. Kluwer Academic Publ., 1996. ISBN 0-7923-9770-3.

[DH97] G. Denker and P. Hartel. TROLL – an object-oriented formal method for distributed information systems design: Syntax and pragmatics. Informatik-Bericht 97-03, Technical University of Braunschweig, 1997.

[EH96] H.-D. Ehrich and P. Hartel. Temporal specification of information systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering*. World Scientific, 1996.

[Ehr96] H.-D. Ehrich. Object Specification. Informatik-Bericht 96-07, Technical University of Braunschweig, 1996.

[EN 87a] CELENEC: Europäische Norm EN 50014. Elektrische Betriebsmittel für explosionsgeschützte Bereiche, Allgemeine Bestimmungen. VDE–Verlag, Berlin, Offenbach, 1987.

[EN 87b] CELENEC: Europäische Norm EN 50018. Elektrische Betriebsmittel für explosionsgeschützte Bereiche, Druckfeste Kapselung "d". VDE–Verlag, Berlin, Offenbach, 1987.

[ES95] H.-D. Ehrich and A. Sernadas. Local specification of distributed families of sequential objects. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types, Joint with the 5th COMPASS Workshop; S. Margherita, Italy*, number 906 in LNCS, pages 219–235. Springer-Verlag, Berlin, May 1995.

[FBGL94] J.S Fitzgerald, T.M Brookes, M.A Green, and P.G Larsen. First results in a comparative study. In M. Naftalin, T. Denvir, and M. Bertrani, editors, *FME'94: Industrial Benefit of Formal Methods*, number 873 in LNCS. Springer-Verlag, Berlin, 1994.

[Gra97] A. Grau. An Animation System for Validating Object-Oriented Conceptual Models. In J.P. Tolvanen and A. Winter, editors, *4th Doctoral Consorcium on Advanced Information Systems Engineering (CAiSE'97), Barcelona*. Fachberichte Informatik 14/97, University Koblenz-Landau, June 1997.

[Har97] P. Hartel. *Konzeptionelle Modellierung von Informationssystemen als verteilte Objektsysteme*. Reihe DISDBIS. infix-Verlag, Sankt Augustin, 1997.

[HDK+97] P. Hartel, G. Denker, M. Kowsari, M. Krone, and H.-D. Ehrich. Information systems modelling with TROLL formal methods at work. *Information Systems*, 22(2-3):79–99, 1997.

[Hoh96] T. Hohnsbein. Objektorientierte Realisierung eines Meßdatenerfassungssystems für druckfeste Kapselung. Diploma thesis, Technical University of Braunschweig, 1996.

[HS94] T. Hohnsbein and H. Schafiee. Reengineering des Programms DRUCKMESS in der PTB. Project work, Technical University of Braunschweig, 1994.

[JWH+94] R. Jungclaus, R.J. Wieringa, P. Hartel, G. Saake, and T. Hartmann. Combining Troll with the Object Modeling Technique. In B. Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen. GI-Fachgespräch FG 1: Integration von semi-formalen und formalen Methoden für die Spezifikation von Software*, Informatik aktuell, pages 35–42. Springer-Verlag, Berlin, 1994.

[KHDE96] M. Kowsari, P. Hartel, G. Denker, and H.-D. Ehrich. A case study in information system design, the CATC system. FME'96: Industrial Benefit and Advances in Formal Methods, Oxford, UK, poster session, March 1996. Available on http://www.cs.tu-bs.de/idb/publications/pub_96.html.

[KKH+96] M. Krone, M. Kowsari, P. Hartel, G. Denker, and H.-D. Ehrich. Developing an information system using Troll – an application field study. In *Conference on Advanced Information Systems Engineering (CAiSE'96), Crete, Greece*, number 1080 in LNCS. Springer-Verlag, Berlin, 1996.

[Kow96] M. Kowsari. Formal object oriented specification language Troll in information system design. In H.-M. Haav and B. Thalheim, editors, *Doctoral Consortium of 2nd International Baltic Workshop on Databases and Information Systems, Tallinn, Estonia*, 1996.

[ORW83] H. Olenik, H. Rentzsch, and W. Wettstein. *Explosion Protection Manual*. W. Girardet, Essen, 2nd revised edition, 1983.

[RBH87] H. Rechenberg, J. Bortfeld, and W. Hanser. *100 Jahre Physikalisch–Technische Bundesanstalt 1887–1987*. VCH Verlagsgesellschaft, Munich, 1987.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object–Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Sch96a] H. Schafiee. Objektorientierte Realisierung der Benutzerschnittstellen eines Meßdatenbearbeitungssystems für druckfeste Kapselung. Diploma thesis, Technical University of Braunschweig, 1996.

[Sch96b] M. Schönhoff. Objektorientierte Realisierung eines Steuerungs- und Überwachungssystems für Explosionsprüfstände. Diploma thesis, Technical University of Braunschweig, 1996. Available on http://www.ifi.unizh.ch/~mschoen.

[WJH+93] R. Wieringa, R. Jungclaus, P. Hartel, T. Hartmann, and G. Saake. omTroll – Object modeling in Troll. In U.W. Lipeck and G. Koschorreck, editors, *International Workshop on Information Systems – Correctness and Reusability (IS-CORE'93), Technical Report No. 01/93, University of Hanover*, pages 267–283, 1993.