D. K. Hammer, J. Hooman, M. A. Reniers, O. van Roosmalen, A. Sintotski

Design of the mine pump control system
1. Introduction

Substantial progress has been made over the past years in the field of real-time control system development. However, the modern development methods, languages, and tools are still not mature enough to solve essential problems in this area. A real-time control system should be described as a reactive system with predictable behavior, including the timing domain. There is, for instance, no development language, method, or tool, which supports the specification, the correctness analysis, and the simple development and maintenance of systems with timing constraints.

End-to-end timing constraints (i.e. the timing constraints between inputs and outputs of a control system) as they appear in a requirements specification, are the natural way to describe timing behavior. Therefore, a good real-time specification language should incorporate an ability to express end-to-end timing constraints. There are, however, few languages that are formal and satisfy this requirement. MSC96 has a formal semantics [Ren99], but it does not incorporate real-time features. The upcoming MSC2000 is a real-time language. The main problem with MSC2000 is that it is only a specification language and it is not intended for the description of an implementation.

The Unified Modeling Language [UML] incorporates Sequence Diagrams, similar to Basic MSCs, and can be used for the specification of real-time systems [Dou98]. UML is extensively supported by tools, for example ARTISAN Real-Time Studio [ARTISAN]. Although end-to-end timing constrains can be specified in terms of Sequence Diagrams, the formal semantics of UML and of Sequence Diagrams in particular is not defined so far. Therefore, the specified timing requirements can only be used for documentation purposes and not for the verification of correctness.

The ROOM development method [SGW94] is used for real-time system development. It uses its own object-based specification and implementation language and has sufficient tool support (Rational Rose RealTime [ROSET], the former ObjecTime Developer). Time is introduced operationally by means of deadlines for firing transitions of state machines and not as end-to-end constraints. The tools provide two modes for execution of an application: simulation and real-time mode. In the real-time execution mode, actions may take time to execute, but in the simulation mode they are executed instantaneously. Thus, it is impossible to introduce a formal semantics for the ROOM specification language that can cover both execution modes.

The ROOM method shows another problem developers struggle with; the implementation of timing behavior can be very complicated. In order to meet the deadlines, a real-time application needs ad-hoc tuning based on changing the priorities of messages. This tuning is not automated and makes the development process relatively complex and expensive. In addition, splitting end-to-end timing constraints at the design phase may introduce an unnecessary and unwanted reduction of the design space.

The problems mentioned above show the lack of an ability to use end-to-end timing constraints throughout the whole development process, coupled with the formal correctness analysis of timing behavior.

The papers [HvR00a, HvR00b] describe an approach for the specification and implementation of real-time systems without splitting end-to-end timing constraints during the design process. According to this method, the system construction process consists of two phases: a platform-independent design phase that includes the specification of timing requirements, and an implementation phase where all platform dependencies are addressed. The second phase differs from a normal compilation only in the sense that also a feasible schedule must be found to execute the program.

The work presented in this report extends the approach from [HvR00a] in two directions. First, a formal semantics for timed message communication is introduced. The formal analysis shows, for example, that the implementation of the mine pump control system presented in [HvR00b] is
incorrect. Second, the simple real-time language from [HvR00a] is extended with an object model. An object model allows specifying a system in terms of objects and interfaces, which makes the development of a complex system more maintainable. Our object model concentrates on information hiding and allows only interface inheritance but not implementation inheritance. An object can thus be considered as a component. Since this report concentrates on the formal specification and verification of systems, no attention is paid to the details of the object (component) model.

Below we explain the three main concepts of our development method: the two-phase application development model, the formal correctness analysis of functional and timed behavior, and composability.

As mentioned before, our application development model distinguishes two phases. In the first phase, the system is decomposed into objects, down to the implementation level. All objects and their interactions are formally specified and verified. In the second phase, the terminal objects are implemented and their resource requirements (processing, memory and input/output) are determined. Based on the properties of the execution platform, a schedule can then be determined. Since resource constraints are only taken into account in the second phase, there is no necessity to split end-to-end timing constraints into artificial pieces, i.e. to define timing constraints per object operation. This also avoids an unnecessary reduction of the design space. Moreover, specification, programming, and formal verification of the real-time system can be done independently of the execution platform. Thus, porting of a real-time application is easier compared with existing approaches.

A formal analysis of the correctness increases the confidence in the correctness of real-time systems. We, therefore, describe objects and their interactions in an assertional way at all levels of refinement. Since this also significantly reduces the testing effort, the time to market for safety-critical systems can be significantly reduced.

Composability is a concept, which allows the construction of a high-level specification from the specification of the constituent components and the properties of the interaction mechanism. This requires that all dependencies between components (including the timing aspects) are made explicit. It is a proven way to struggle with the complexity of software systems, especially in the world of object-oriented software development. It simplifies the maintainability of the system and separates the verification of a component from the verification of its use in a particular context. This makes formal verification more feasible for real-world systems.

This work concentrates on a compositional framework that combines the specification and verification of functional requirements and end-to-end timing constraints into one consistent formal model. In this report, we apply the approach to the mine pump control system. The formal analysis shows that a previously published implementation [HvR00b] of the mine pump control system is incorrect. We also present a decomposition of the mine pump control system that is formally proven to be correct.

The correctness of the decomposition was checked using the PVS proof checker [OSR92, PVS]. PVS is an interactive proof checker using higher-order predicate logic. Appendix B gives brief overview of PVS. The complete PVS sources of the presented example can be found in [DUMP].

The report is organized as follows. Section 2 contains an overview of the object model and the formal semantics of communications mechanisms. Section 3 illustrates how the presented framework can be used for design of a real-time system. The concluding remarks can be found in Section 4. Appendix A contains a list of used symbols and appendix B contains a PVS overview.
2. Tools overview

In this chapter we discuss two important classes of tools for supporting formal methods: real-time model checkers and theorem provers.

2.1. Real-time model checking

Model checking is a technique that relies on building a model of a system and checking that a desired property holds in that model. The check is performed as an exhaustive search over the model's state space. Usually, a system is modeled as a finite labeled transition system and, therefore, this search always terminates. In many cases, the desired property is formulated as a temporal logic formula [Pnueli81]. However, some of the model checking systems [CPS93, FGK96] accept automata as a desired property specification. Vardi and Wolper [VW86] showed that temporal-logic model checking could be performed as automata-based model checking, so both types of model checking are closely related.

There are several successful implementations of model checking that can be used for the verification of real-time systems: Kronos [BTY97], UPPAAL [LPY97], and HyTech [AHH96, HHT97].

Kronos [BTY97] is a verification-support tool based on timed automata and temporal logic. A timed automaton [AD94] is a finite state machine extended with a finite set of real-valued variables, called clocks, used to express timing constraints. Each timed automaton may have as many clocks as needed. Clocks can be reset to zero and their values increase uniformly with time. At any instant, the value of a clock is equal to the time elapsed since the last time it was reset. A transition is enabled only if the timing constraint associated with it is satisfied by the current values of the clocks. Timing constraints such as propagation delays, execution times, and response times, are expressed as predicates on the values of the clocks.

A timed automaton models the behavior of a single process of the system. A complex real-time system made up of several cooperating and communicating processes is modeled as the parallel composition of the corresponding timed automata. To model inter-process communication, the transitions of the timed automata are labeled with sets of identifiers interpreted as synchronization channels. Correctness criteria can be specified in two different ways: (1) as formulas of the timed temporal logic TCTL [ACD93], an extension of the temporal logic CTL that allows quantitative temporal reasoning over dense time, or (2) as timed automata.

UPPAAL [LPY97] is a toolset for modeling, specification, and verification of real-time systems. Just as in Kronos, a network of timed automata models the system behavior. To provide a more expressive model, the timed automata formalism was extended with variables of various datatypes such as boolean and integer. Automata may communicate either via global integer variables or using communication channels. A communication channel is used for synchronization of two objects and its semantics is defined just as in CCS [Milner89]. The desired properties of the system can be specified either using the temporal real-time logic from [LPY95], or by a timed automaton.

Although HyTech [AHH96, HHT97] is aimed for the verification of hybrid systems, it can also be used for verification of discrete real-time systems. It uses linear hybrid automata for system models and branching-time temporal logic [CE81] for specifications. A hybrid automaton [ACH93] is a timed automaton extended by a finite number of real-valued variables that change continuously, as specified by differential equations and differential inequalities. In order to make the correctness analysis completely automatic, the hybrid automata are restricted in HyTech to the linear hybrid automata [AHH96]. In a linear hybrid automaton, the dynamics of the continuous variables are defined by linear differential inequalities. Since the number of possible constraints is no longer finite, when moving from timed automata to linear hybrid automata, the price to pay for the increased generality is the loss of guaranteed termination for model checking. The method is still of practical interest, however, because termination happens naturally in many examples.
HyTech can be used to provide more than a mere “yes” or “no” answer to the question of whether a system satisfies a correctness requirement. If a system description contains design parameters, whose values are not specified, HyTech computes necessary and sufficient constraints on the parameter values that guarantee correctness, and thus, can be used for the parametric synthesis of real-time systems.

To sum up, several reasons make model checking a very attractive verification technique. First, it is a completely automatic approach, which allows to verify systems very fast, sometimes in matter of minutes. Second, if the verification fails, it can produce counterexamples showing the errors in the design of the system being verified and, so, it can be used to aid debugging. Third, some properties (for example, absence of deadlocks in a system) can be analyzed automatically even without specifying the properties explicitly by the user.

However, the use of model checkers may be difficult. The most obvious problem is the state explosion problem. Techniques like Binary Decision Diagrams (BDD) [Bryant86] and various order reduction techniques [Kursh94a, Kursh94b, Peled96] help to represent states efficiently, thereby increasing the size of the systems that could be verified. The modeling languages used in model checkers often support synchronous communication and shared variables only and, therefore, they are not particularly attractive for modeling distributed systems.

Finally, a drawback of model checking regarding end-to-end timing constraints is that the model has to be operational enough in order to be verified against the specification of desired properties by a model checker. Programs with end-to-end timing constraints require scheduling in order to be executed but verification of the programs after such scheduling is pointless in our approach because such a verification is not platform-independent anymore.

2.2. Theorem provers

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in the same mathematical logic. This logic is defined by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas.

Theorem proving has several advantages compared to model checking. Theorem provers can deal with infinite state spaces. They rely on techniques like induction to perform proofs over infinite domains. Theorem provers normally have no restrictions on the semantics of program statements and on the used specification method. This generality is achieved for the price of less automation.

Theorem provers can be roughly classified in a spectrum from highly automated programs, very effective in particular domains, to interactive systems.

Automatic theorem provers, such as Nqthm [BM79], its successor ACL2 [KM95], Eves [CKM88], LP [GG88], Reve [Lesc83], RRL [KM87], are guided by a sequence of lemmas and definitions. Each theorem is proved automatically using built-in heuristics for induction, instantiation, lemma-driven rewriting, and simplification. The success of theorem provers is based on the fact that some techniques work remarkably well on specialized problem domains. The downside of automatic theorem provers is that they still require human interaction, which is really about proving but not about proof itself. Such interactions may include ordering of variables in formulae, the invention and ordering of lemmas, experimenting and so on. Another disadvantage is that after use of such a theorem prover, the user knows more about the prover but not about the stated problem. This holds for model checking as well.

Interactive theorem provers, such as HOL [Gordon87], Isabelle [Paul94], Coq [DFH93], PVS [OSR92, PVS], lack much automation and require a skilled person to operate them. However, these tools provide mechanical verification support for the broadest range of the verification problems. Since the user has complete control over the proof, the user can learn much about the
system being verified. These facts, and the fact that the PVS theorem prover has been used to verify a large number of real-time and fault-tolerant algorithms [CM97, Hoo94a, Hoo95, HV96, Hoo96, AH96a, AH96b, AH97] played important roles in choosing the PVS theorem prover to be the mechanical verification support tool for this project.

2.3. Summary

The following table gives an overview of the features of model checking and theorem proving tools.

<table>
<thead>
<tr>
<th></th>
<th>Model checking</th>
<th>Theorem proving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree of automation</td>
<td>Fully automatic</td>
<td>Mechanized</td>
</tr>
<tr>
<td>Number of states</td>
<td>Finite</td>
<td>Possibly infinite</td>
</tr>
<tr>
<td>Communication between concurrent processes</td>
<td>Synchronous, Asynchronous with finite buffers</td>
<td>No restrictions</td>
</tr>
<tr>
<td>Addition of end-to-end timing constraints</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
<tr>
<td>Compositional reasoning</td>
<td>No</td>
<td>Possible</td>
</tr>
<tr>
<td>Hierarchical system decomposition</td>
<td>No (in the UPPAAL “to do” list)</td>
<td>Possible</td>
</tr>
</tbody>
</table>
3. Framework overview

In this section, we present an overview of the object model, the formal semantics of the communication mechanism, and the overview of the specification technique.

3.1. Object model

Our object model consists of interfaces, composite objects and terminal objects. This section describes the meaning of these terms and their relations.

An object is a system construction unit. An object may be terminal or composite. Terminal objects have a sequential nature. A composite object is a parallel composition of a set of intercommunicating objects, which are called contained. Contained objects, in their turn, may be terminal or composite again. This means that the systems under development can be presented as a hierarchy of objects where composite and terminal objects play the roles of nodes and leaves respectively. A system under construction itself is an object, communicating with the environment.

Object development is class-based. I.e. it is possible to define a class of objects with similar behavior and instantiate this class many times as objects in the system under development. In order to support strong encapsulation, only interface inheritance is allowed but not implementation inheritance.

Each object, composite or terminal, communicates with its environment via interfaces. Interfaces hide implementation details of objects and support the encapsulation mechanism. Thus, objects look like black boxes with interfaces from the outside world and an external observer cannot distinguish between composite and terminal objects that support similar interfaces. Each object may implement as many interfaces as necessary.

An interface is an abstraction of the expected object’s behavior. It does not contain any implementation details but its high-level specification only. An interface is not tightly connected to a particular object: different objects may implement the same interface.

Each interface contains two parts, a structural and a behavioral one. The first part describes the interface elements while the second part specifies expected object’s behavior in terms of communication acts through the interface elements, described in the structural part. The structural part of an interface is also called the interface signature while the behavioral part is called the interface specification.

The object model as well as its graphical notation is based on the ROOM approach [SGW94] and has some resemblance with the HOOD method [BW95] as well. The ROOM object model has several advantages in the context of real-time systems:

1) It focuses on objects and communication between them rather than on classes and associations between classes. Since most real-time applications have a static structure, it is convenient to see the system under development as a set of intercommunicating objects rather than as a set of classes with associations. Object-based models are easier to analyze with regard to their execution behavior and timeliness properties.

2) In a system consisting of concurrent objects, another advantage is the replacement of function calls by message passing since the invariants describing the properties of objects may be violated during a function call. For instance, if an object calls a function of another object and this function calls the first object again in order to complete the calculation (the reentrance situation), the behavior specification of this system can be very confusing and unclear. Using message passing, a reentrance situation never takes place.

3) The ROOM object model is based on an asynchronous communication mechanism and is easy to adapt to the simple programming language described in [HvR00a].
Another important advantage of using the ROOM object model and notation is that it probably becomes a standard notation for the object-oriented modeling of real-time applications. The UML-RT [ROSERT], extension of UML is based on ROOM, a prominent proposal for certification by the Object Management Group. Moreover, UML-RT is used in the real-time version of the Rational Rose environment, Rational Rose RealTime.

Although our object model follows that of ROOM very closely, it has some differences. The most important ones are:

- An object is not a combination of reactive behavior and encapsulated intercommunicating objects, executing in parallel with this reactive behavior. Each object is either composite or terminal. The behavior of a composite object is specified by the parallel composition of intercommunicating objects while the behavior of a terminal object is specified as a sequential composition of statements. This choice is made for simplicity. It does not reduce the expressive power of the formalism.

- An object, composite or terminal, implements one or many interfaces, which are abstractions of objects. Our object model does not support object inheritance (actors in ROOM). It supports, however, multiple inheritance of interfaces. This, again, does not reduce the expressive power of the language but significantly simplifies the object model.

3.2. Communication model

3.2.1. Overview of the communication model

Objects use two communication mechanisms: message passing and device registers. Each object may have several input ports, output ports, and device registers.

Communication via device registers is an asynchronous communication by means of a non-blocking buffer of size one. A device register may change its value continuously over time and can be used to model a continuously changing environment of a control system. Specification of such a control system by means of device registers makes the formal specification closer to the requirements specification. At any time, an object may read the current value of the device register or change it by writing a new value. A detailed description and the formal semantics of communication by means of device registers can be found in section 2.2.4.

Message passing is a unidirectional communication mechanism via reliable FIFO channels. Like device registers, the message passing is an asynchronous communication mechanism, i.e. a sender of a message never waits for the readiness of a receiver and always continues its execution after sending a message. Although a synchronous communication (when a sender always waits for a receiver to be ready) usually has a simpler formal semantics, there are several reasons to use message passing as a communication mechanism in distributed real-time systems. One reason is that synchronous communication may reduce performance because it may lead to needless blocking of objects. This blocking is also dangerous in a distributed environment in case of unreliable communication channels. In our communication model, channels are reliable but usage of message passing gives the opportunity to extend the current communication model with unreliable channels.

The communication by message passing works as follows. An object may send a message to an output port. Sending of a message takes some time but always finishes within a finite time interval. After some time, the message sent is arriving at the input port connected to the output port, provided that such an input port exists. After this moment the message is ready to be consumed by the object that owns the input port according to a FIFO ordering policy. Arrival of a message at an input port is called “delivery of a message” and consumption of a message by an object is called “acceptance of a message” in the rest of the report.
The asynchronous behavior of message passing can be represented in two ways in the formal model. First, an asynchronous channel can be explicitly placed between the interfaces of communicating objects (Figure 1).

![Diagram of an "external" asynchronous channel](image)

Figure 1. An “external” asynchronous channel.

The second possibility is hiding the asynchronous communication channels behind the interface of receiving objects. In this case, a connection between objects is nothing more than a mapping of port names (Figure 2).

![Diagram of an "internal" asynchronous channel](image)

Figure 2. An “internal” asynchronous channel.

The latter case is better from the formal verification point of view. In the first case, the verification of a composite object always includes the usage of the message passing semantics. The verification may be quite complex for a large composite object, because the message passing semantics is not trivial and must be used in order to describe the connections between inner objects formally. If an asynchronous channel is hidden inside a receiver object, the verification is much easier because connections between inner objects can be described by simple equations. Although the verification of a terminal object (where the asynchronous channel appears) becomes more complex with this trick, it does not require great additional verification effort. Therefore, the latter possibility was chosen.

### 3.2.2. Common definitions

Time is continuous and the time domain \( \text{TIME} \) can be described as the set of non-negative real numbers: \( \text{TIME} = \{ t \in \mathbb{R} | t \geq 0 \} \).

A timed predicate is a convenient notion for dealing with timed events. The set of timed predicates is defined as \( \mathcal{P} = \{ P | P : \text{TIME} \rightarrow \mathbb{B} \} \), where \( \mathbb{B} \) is the set of boolean values. Negation, conjunction, and disjunction are defined for the timed predicates. For example, a conjunction of two timed predicates \( p_1 \) and \( p_2 \) is a timed predicate, which is defined as follows: \( (p_1 \land p_2)(t) \equiv p_1(t) \land p_2(t) \). Negation and disjunction are defined similarly.

The following two abbreviations are used throughout the report:

\[
\forall P \in \mathcal{P}, I \subseteq \text{TIME} : P \text{ during } I \equiv \forall t : t \in I \rightarrow P(t) \\
\forall P \in \mathcal{P}, I \subseteq \text{TIME} : P \text{ inside } I \equiv \exists t : t \in I \land P(t).
\]
PVS implementation

Timed predicates are specified in PVS in two steps: first, the abstract operations on predicates are defined in the theory “predicates” and, second, the theory “time” defines time and timed predicates.

Theory “predicates”

The theory defines operations for predicates on type, specified as “T”.

```
predicates[ T: TYPE ]; THEORY

BEGIN

  P, Q : VAR PREDICATE[ T ];
  t  : VAR T;

The following operations are defined for predicates:

  TRUE(t) : bool = TRUE;
  FALSE(t): bool = FALSE;
  NOT(P)(t) : bool = NOT P( t );
  AND(P,Q)(t) : bool = P(t) AND Q(t);
  OR(P,Q)(t) : bool = P(t) OR Q(t) ;
  IMPLIES(P,Q)(t): bool = P(t) IMPLIES Q(t) ;
  IFF(P,Q)(t) : bool = P(t) IFF Q(t);

END predicates
```

Theory “time”

The theory defines time and introduces notions of interval, timed predicate and some useful abbreviations.

```
time: THEORY

BEGIN

Type “Time” is a set of nonnegative real numbers

  Time: TYPE = nonneg_real;

IMPORTING predicates[ Time ]

Time interval is a set of time moments:

  TimeInterval : TYPE = SETOF[ Time ];

Timed predicate is a predicate defined on time values:

  TimePredicate: TYPE = PREDICATE[ Time ];

The theory introduces abbreviations for dense open and closed time intervals:

  cct( u, v: Time ): TimeInterval = ( t | u <= t AND t <= v );
  co( u, v: Time ): TimeInterval = ( t | u <= t AND t < v );
  oct u, v: Time ) : TimeInterval = ( t | u < t AND t <= v );
  oo( u, v: Time ) : TimeInterval = ( t | u < t AND t < v );
```
The following abbreviations implements predicates \( P \) inside \( I \) and \( P \) during \( I \) in PVS:

\[
\begin{align*}
\text{inside}(P: \text{TimePredicate}, I: \text{TimeInterval}) : \text{bool} &= \exists t : \text{Time} . \text{member}(t, I) \land P(t); \\
\text{during}(P: \text{TimePredicate}, I: \text{TimeInterval}) : \text{bool} &= \forall t : \text{Time} . \text{member}(t, I) \implies P(t);
\end{align*}
\]

\text{END time}

3.2.3. The formal semantics of the message passing

We use PORTS and \( D \) as notations for, respectively, the set of ports and the set of data values that can be transferred by messages.

The sending of a data value \( v \in D \) to a port \( p \) at a moment in time \( t \) is expressed as \( p(v)(t) \).

More formally, each port is represented by a function, which transforms transferred data into a timed predicate: \( p : D \to P \).

A terminal object on the other side of a communication channel may accept a message, which has been sent to its port. This acceptance event is modeled by a function, which given the input port, a transferred data value, and a sending time leads to a timed predicate:

\[
\text{accept} : \text{PORTS} \times D \times \text{TIME} \to P.
\]

The expression \( \text{accept}(p,v,t_a)(t_a) \) holds true if and only if the object accepts the data value \( v \) (sent at time \( t_a \)) at the port \( p \) at the moment \( t_a \). The value \( t_a \) is used for the unique identification of a message in the channel, because different messages may transfer the same data value.

The following abbreviations are used throughout the report:

\[
\begin{align*}
\text{accept}(p,v)(t_a) &\equiv \exists t : \text{accept}(p,v,t)(t), \\
\text{accept}(p)(t_a) &\equiv \exists v : \text{accept}(p,v)(t_a).
\end{align*}
\]

The following axioms define the semantics of message passing.

Axiom P1 (Unique Delivering)

Only one data value can be sent (and delivered) with a message at the same port at the same time:

\[
\forall p \in \text{PORTS}, v_1, v_2 \in D, t \in \text{TIME} : (p(v_1)(t) \land p(v_2)(t)) \to v_1 = v_2.
\]

\( \square \)

Axiom P2 (Unique Accepting)

Only one message can be accepted at the same port at the same time:

\[
\forall p \in \text{PORTS}, v_1, v_2 \in D, t, t'_1, t'_2 \in \text{TIME} : \\
\quad \text{accept}(p,v_1,t_1)(t_a) \land \text{accept}(p,v_2,t_2')(t_a) \to v_1 = v_2 \land t'_1 = t_2'.
\]

\( \square \)
Axiom P3 (Accepting Once)
The moment of delivery allows the unique identification of the acceptance of a message:

$$\forall p \in \text{PORTS}, v \in D, t_s, t'_s, t''_s \in \text{TIME} :$$

$$\text{accept}(p, v, t_s)(t'_s) \land \text{accept}(p, v, t_s)(t''_s) \rightarrow t''_s = t'_s.$$  

□

Axiom P4 (Causality)
Every accepted message has been delivered beforehand:

$$\forall p \in \text{PORTS}, v \in D, t_s, t_a \in \text{TIME} : \text{accept}(p, v, t_s)(t_a) \rightarrow p(v)(t_s) \land t_a < t_s.$$  

Note: This axiom states that there is some time between the delivery moment and the acceptance moment, i.e. the delivery and the acceptance of a message cannot happen at the same moment. We could use $t_s \leq t_a$ instead of $t_a < t_s$ but it complicates other parts of the message passing semantics, which are not presented here because they have not been used for the correctness analysis of the mine pump case study.

□

Axiom P5 (FIFO)
Messages from the same input port are consumed by the destination object in “first in – first out” order:

$$\forall p \in \text{PORTS}, v_1, v_2 \in D, t'_1, t''_1, t'_2, t''_2 \in \text{TIME} :$$

$$\text{accept}(p, v_1, t'_1)(t''_1) \land \text{accept}(p, v_2, t'_2)(t''_2) \rightarrow (t''_1 < t''_2 \iff t'_1 < t'_2).$$  

□

Axiom P6 (Last Accepted Message)
There is a last acceptance of a message within every closed time interval if an acceptance ever took place within that interval:

$$\forall p \in \text{PORTS}, t_s, t_a \in \text{TIME} : \text{accept}(p) \text{ inside } [t_s, t_a] \rightarrow$$

$$\exists t_a \in [t_s, t_a] : \text{accept}(p)(t_a) \land (\neg \text{accept}(p) \text{ during } (t_s, t_a)).$$  

Note: This axiom forbids Zeno behavior of the object accepting messages.

□

PVS implementation
The listed axioms defining the semantics of message passing are specified in the theory “port”. This theory has a parameter “Data”, which is a non-empty type of data transferred with messages.

```pvs
port[
    Data: TYPE+
] : THEORY
BEGIN
IMPORTING time;

Port: TYPE = FUNCTION[ Data -> TimePredicate ];
accept: FUNCTION[ Port, Data, Time -> TimePredicate ];
```
The following PVS code defines the useful abbreviations for “accept”:

\[
\begin{align*}
\text{accept}(p, x) & : \text{TimePredicate} = \text{LAMBDA } t: \\
& \quad \text{EXISTS } td: \text{accept}(p, x, td)(t); \\
\text{accept}(p) & : \text{TimePredicate} = \text{LAMBDA } t: \\
& \quad \text{EXISTS } x, td: \text{accept}(p, x, td)(t);
\end{align*}
\]

The following group of axioms, which is called “Uniqueness axioms”, specifies P1 (Unique Delivering), P2 (Unique Accepting), and P3 (Accepting Once) respectively:

\[
\begin{align*}
\text{pUniqueDeliver: AXIOM} \\
& \quad \text{FORALL } p, x_1, x_2, td: \\
& \quad \quad p(x_1)(td) \land p(x_2)(td) \implies x_1 = x_2;
\end{align*}
\]

\[
\begin{align*}
\text{pUniqueAccept: AXIOM} \\
& \quad \text{FORALL } p, x_1, x_2, td_1, td_2, ta: \\
& \quad \quad \text{accept}(p, x_1, td_1)(ta) \land \\
& \quad \quad \text{accept}(p, x_2, td_2)(ta) \implies x_1 = x_2 \land td_1 = td_2;
\end{align*}
\]

\[
\begin{align*}
\text{pAcceptOnce: AXIOM} \\
& \quad \text{FORALL } p, x, td, ta_1, ta_2: \\
& \quad \quad \text{accept}(p, x, td)(ta_1) \land \\
& \quad \quad \text{accept}(p, x, td)(ta_2) \implies ta_1 = ta_2;
\end{align*}
\]

P4 (Causality) is specified in PVS as follows:

\[
\begin{align*}
\text{pCausality: AXIOM} \\
& \quad \text{FORALL } p, x, td, ta: \\
& \quad \quad \text{accept}(p, x, td)(ta) \implies p(x)(td) \land td < ta;
\end{align*}
\]

The next axiom is P5 (FIFO):

\[
\begin{align*}
\text{pFIFO: AXIOM} \\
& \quad \text{FORALL } x_1, x_2, td_1, td_2, ta_1, ta_2: \\
& \quad \quad \text{accept}(p, x_1, td_1)(ta_1) \land \\
& \quad \quad \text{accept}(p, x_2, td_2)(ta_2) \implies (td_1 < td_2 \iff ta_1 < ta_2);
\end{align*}
\]

The last axiom in the theory is P5 (Last Accepted Message):

\[
\begin{align*}
\text{pLastAccepted: AXIOM} \\
& \quad \text{FORALL } p, t_1, t_2: \\
& \quad \quad \text{inside}(\text{accept}(p), \text{cc}(t_1,t_2)) \implies \\
& \quad \quad \text{EXISTS } t: \text{member}(t, \text{cc}(t_1,t_2)) \land \\
& \quad \quad \text{accept}(p)(t) \land \\
& \quad \quad \text{during}(\text{NOT accept}(p), \text{oc}(t,t_2));
\end{align*}
\]

END port
3.2.4. The formal semantics of the device registers

We use DR as a notation for the set of device registers. Each device register can be presented as an entity, which stores some data value that can change continuously in time:

\[ \text{DR} \equiv \{ d \mid d : \text{TIME} \to D \} . \]

The functions read and write define the notions of read and write operations:

\[
\begin{align*}
\text{read} : \text{DR} \times D & \to P, \\
\text{write} : \text{DR} \times D & \to P .
\end{align*}
\]

Informally, the semantics of these functions can be explained as follows. The expression \( \text{read}(d,v)(t) \), where \( d \in \text{DR}, v \in D, t \in \text{TIME} \), holds true if and only if the value \( v \) was read from the device register \( d \) at time \( t \). The expression \( \text{write}(d,v)(t) \), where \( d \in \text{DR}, v \in D, t \in \text{TIME} \), holds true if and only if the value \( v \) was written to the device register \( d \) at time \( t \). The formal meaning of these functions is presented below.

The following abbreviations will be used:

\[
\begin{align*}
\text{read}(d)(t) & \equiv \exists v \in D : \text{read}(d,v)(t) , \\
\text{write}(d)(t) & \equiv \exists v \in D : \text{write}(d,v)(t) .
\end{align*}
\]

The following axioms define the semantics of device registers.

Axiom DR1 (Reading Value)

The reading of a device register is the reading of the value that is stored in the corresponding device register at a time of reading:

\[
\forall d \in \text{DR}, v \in D, t \in \text{TIME} : \text{read}(d,v)(t) \to d(t) = v .
\]

Note: Using axioms DR1 and DR2, one can see that if the device register is being reading and writing at the same moment, the read value is a new value written to the device register at this moment.

Axiom DR2 (Writing Value)

If a device register holds some value, this value either the initial value or the value last written to the device register:

\[
\forall d \in \text{DR}, v \in D, t \in \text{TIME} : \\
\begin{align*}
d(t) & = v \iff (\exists t_0 \leq t : \text{write}(d,v)(t_0) \land (\neg \text{write}(d) \text{ during } (t_0,t))) \lor \\
& (d(0) = v \land (\neg \text{write}(d) \text{ during } [0,t])).
\end{align*}
\]

Note: Using axioms DR1 and DR2, one can see that if the device register is being reading and writing at the same moment, the read value is a new value written to the device register at this moment.

Axiom DR3 (Unique Writing)

Only one value can be written to the device register at the same time:

\[
\forall d \in \text{DR}, t \in \text{TIME}, v_1, v_2 \in D : \text{write}(d,v_1)(t) \land \text{write}(d,v_2)(t) \to v_1 = v_2 .
\]
Axiom DR4 (Finiteness)
Only allow a finite number of writing actions in a finite amount of time:
\[ \forall d \in \text{DR,} I \subseteq \text{TIME} : \]
\[ \text{write}(d) \text{ inside } I \rightarrow \exists t \in I : \text{write}(d)(t) \land (\neg \text{write}(d) \text{ during } \{ t_0 \in I \mid t_0 < t \}) , \]
provided that \( I \) is a finite interval.

Lemma DR5 (Discrete Write)
This useful lemma states that a device register keeps some value during an interval if this value is written to the device register before the interval's opening and no other value is written to it before the interval's ending:
\[ \forall t_1,t_2 \in \text{TIME}, v \in D : \]
\[ (\exists w \leq t_1 : \text{write}(d,w)(t_1) \land \forall w \in D : w \neq v \rightarrow \neg \text{write}(d,w) \text{ during } (t_1,t_2)) \rightarrow \]
\[ (d = v) \text{ during } [t_1,t_2] . \]

PVS implementation

The following PVS theory, "deviceRegister", defines device registers, which store the value of a non-empty type "Data", along with their properties:

deviceRegister[
  Data: TYPE+
]: THEORY

BEGIN

% Importing necessary theories

IMPORTING time;

The following type defines the set of device registers:

DeviceRegister: TYPE = FUNCTION[ Time -> Data ];

% Variables declaration

d: VAR DeviceRegister;
v, w, v1, v2: VAR Data;
t, t0, t1, t2, tw: VAR Time;
I: VAR TimeInterval;

The "read" operation and its abbreviation are defined as follows:

read: FUNCTION[ DeviceRegister, Data -> TimePredicate ];

read( d ) : TimePredicate = LAMBDA t:
  EXISTS v: read( d, v )( t );
The meaning of the “read” operation is defined by the axiom DR1 (Reading Value), which is specified in the PVS implementation as the following:

\[
\text{drReadValue: AXIOM} \\
\text{read}(d,v)(t) \text{ IMPLIES } d(t)=v; \\
\]

The “write” operation and its abbreviation are defined as follows:

\[
\text{write: FUNCTION[ DeviceRegister, Data } \rightarrow \text{ TimePredicate ];} \\
\text{write( d ): TimePredicate = LAMBDA t:} \\
\text{EXISTS v: write( d, v )}( t ); \\
\]

The following axioms are PVS implementations of DR2 (Writing Value) and DR3 (Unique Writing) respectively:

\[
\text{drWriteValue: AXIOM} \\
\text{d(t)=v IFF } ( \\
\quad \text{EXISTS t0: t0<=t AND} \\
\quad \text{write}(d,v)(t0) \text{ AND} \\
\quad \text{FORALL w: w /= v =>} \\
\quad \quad \text{during( NOT write}(d,w), \text{ oc}(t0,t) ) \text{ OR } ( \\
\quad \quad \quad v=d(0) \text{ AND} \\
\quad \quad \quad \text{FORALL w: w /= v =>} \\
\quad \quad \quad \quad \text{during( NOT write}(d,w), \text{ cc}(0,t) ) \text{ )} \text{ OR } ( \\
\quad \quad \quad v=d(0) \text{ AND} \\
\quad \quad \quad \text{FORALL w: w /= v =>} \\
\quad \quad \quad \quad \text{during( NOT write}(d,w), \text{ cc}(0,t) ) \text{ )} \text{ ;} \\
\text{drUniqueness: AXIOM} \\
\text{write}(d,v1)(t) \text{ AND write}(d,v2)(t) \text{ IMPLIES v1=v2; } \\
\]

The following formula defines the finite interval as an interval, which is bounded from above:

\[
\text{drFinite}(I): \text{ bool = EXISTS t1: FORALL t: member(t,I) => t<}t1; \\
\]

The axiom DR4 (Finiteness) and lemma DR5 (Discrete Write):

\[
\text{drFiniteness: AXIOM} \\
\quad \text{FORALL I, t1, t2: drFinite}(I) \text{ IMPLIES (} \\
\quad \quad \text{inside}( \text{ write}(d), \text{ I } ) \text{ IMPLIES} \\
\quad \quad \quad \text{EXISTS t: member( t, I ) AND} \\
\quad \quad \quad \text{write}(d)(t) \text{ AND} \\
\quad \quad \quad \text{FORALL t0: t0 > t =>} \\
\quad \quad \quad \quad \text{NOT write}(d)(t0) \text{ )} \text{ ;} \\
\text{drDiscreteWrite: LEMMA} \\
\quad \text{FORALL t1, t2:} \\
\quad \quad ( \\
\quad \quad \quad \text{EXISTS tw: tw <= t1 AND} \\
\quad \quad \quad \text{write}( d, v )( \text{ tw }) \text{ AND} \\
\quad \quad \quad \text{FORALL w: w /= v =>} \\
\quad \quad \quad \quad \text{during( NOT write}(d,w), \text{ oc}(t\text{w},t2) ) \text{ )} \text{ IMPLIES} \\
\quad \quad \quad \text{during( ( LAMBDA t: d(t) = v ), cc(t1,t2) )}; \\
\]

END deviceRegister
3.3. The specification technique

Any interface consists of two parts: a structural part and a behavioral part. The former contains the description of device registers, input and output ports used for communication with the environment. The latter specifies the behavior of the objects that implement the interface in terms of input and output events related to the interface. The technique described in [HvR00a] is used for the behavior specification. According to this technique, each interface specification has the form \( \langle (A) \rangle I \langle (C) \rangle \), where \( I \) is the identifier of the specified interface, \( A \) is an assertion called assumption, and \( C \) is an assertion called commitment. An implementation of interface \( I \) is correct if \( C \) holds during and after its execution provided that \( A \) holds before the execution.
4. Design of the mine pump control system

In this section, we present the formally driven design of a real-time control system. The top-level specification of the mine-pump control system is introduced in section 4.1. Sections 4.2 and 4.3 describe transformations of the top-level specification into a specification that is easier to implement. Section 4.4 describes the decomposition of the control system and specifications of its components. The problem of temporal inaccuracy is discussed in section 4.5. Section 4.6 contains solution for the temporal accuracy problem and the correctness analysis of new decomposition.

4.1. The top-level specification of the mine pump control system

The paper [HvR00b] suggests a solution of the mine pump problem [BL91] using a simple real-time programming language. However, no formal analysis of the suggested solution was performed and, therefore, its correctness is questionable. In this section, we will present a formally developed decomposition of the mine pump control system, which shows that the implementation of the mine pump control system presented in [HvR00b] is indeed incorrect.

In this report, we consider a simplified version of the mine pump problem. The simplification is based on the assumption that the pump never fails and, therefore, the water flow sensor is unnecessary and can be omitted. Furthermore, in order to make the formal specification and verification easier, we use a slightly different decomposition of the system. However, the changes we made are not critical for the correctness of the system and they do not affect the problem detected in the implementation presented in [HvR00b].

The following informal description of the control system is based on [HvR00a]. The mine pump control system controls a vessel, which has a limited but unknown influx of water. A pump with a larger capacity than this influx is installed to be able to remove the water from the vessel. Activation or deactivation of the pump must prevent overflowing or running dry of the vessel. A special device measures the water level. The control system should switch the pump on or off within certain deadlines, which are calculated from the maximal flow in and out of the vessel and the selected critical levels. The values $L$ and $H$ denote the critical low and high levels of water, and the values $DL$ and $DH$ denote the corresponding deadlines. The control system should switch the pump off in case the methane level around the physical pump exceeds a critical value $M$ and switch it on if the methane level is below the critical value. Similarly, a maximum time interval of length $DC$ for switching the pump off in dangerous situations can be calculated from the parameters of the environment.

The mine pump control system can be presented as an object, implementing the interface $IMinePump$ as shown in Figure 3:

![Figure 3. The interface $IMinePump$.](image)

This interface comprises three device registers. The input device registers $water$ and $methane$ keep values of the current measurements of the water and methane levels. The output device register $pump$ of enumeration type $\{on, off\}$ reflects the state of the physical pump.

According to the informal description given above, the following conditions on the environment are relevant for the control system.

23
The part of the specification “the water level is above the critical value $H$” means that the instant value of the device register \textit{water} is greater than or equal to $H$. This condition is expressed by the abbreviation $\text{WaterHigh}(t) \equiv \text{water}(t) \geq H$. Similarly, the condition that the water level is below the critical value $L$ is formulated as $\text{WaterLow}(t) \equiv \text{water}(t) \leq L$.

In the same way, the part of the specification “the methane level is above the critical value $M$” means that the instant value of the device register \textit{methane} is greater than or equal to $M$. This condition is denoted as $\text{Danger}(t) \equiv \text{methane}(t) \geq M$. The situation where the methane level is not dangerous is denoted as $\neg \text{Danger}(t)$.

Using these expressions, the informal specification of the mine pump system can be formalized as follows.

The control system should keep the pump switched on when the water level is high and the methane level is below its critical value. The deadline for the activation of the pump is $DH$. The same pump activation deadline $DH$ holds if the water level is high and the methane level becomes safe again:

\[
\text{sysWaterHigh} \equiv \forall t_1,t_2 : (\text{WaterHigh} \land \neg \text{Danger}) \text{ during } [t_1,t_2] \rightarrow (\text{pump} = \text{on}) \text{ during } [t_1 + DH,t_2] \]

The following part of the specification states that the control system should keep the mine pump switched off if the water level is below its critical level. The deadline for deactivation of the pump is $DL$:

\[
\text{sysWaterLow} \equiv \forall t_1,t_2 : \text{WaterLow} \text{ during } [t_1,t_2] \rightarrow (\text{pump} = \text{off}) \text{ during } [t_1 + DL,t_2] .
\]

Finally, the last part of the top-level specification states that the control system should keep the mine pump switched off if the methane level is above critical. The deadline for deactivation of the pump in this case is $DC$:

\[
\text{sysDanger} \equiv \forall t_1,t_2 : \text{Danger} \text{ during } [t_1,t_2] \rightarrow (\text{pump} = \text{off}) \text{ during } [t_1 + DC,t_2] .
\]

Since the three assertions mentioned above have the same structure, we use the following generalization:

\[
\text{sysBehavior}(C,A,d) \equiv \forall t_1,t_2 : C \text{ during } [t_1,t_2] \rightarrow A \text{ during } [t_1 + d,t_2] .
\]

The timed predicate $C$ here is a condition on the environment, $A$ is a timed predicate specifying output of the control system, and $d$ is a deadline.

Using this generalized predicate, the assertions may be rewritten as follows:

\[
\text{sysWaterHigh} \equiv \text{sysBehavior}(\text{WaterHigh} \land \neg \text{Danger}, \text{PumpIsOn}, DH) ;
\]
\[
\text{sysWaterLow} \equiv \text{sysBehavior}(\text{WaterLow}, \text{PumpIsOff}, DL) ;
\]
\[
\text{sysDanger} \equiv \text{sysBehavior}(\text{Danger}, \text{PumpIsOff}, DC) ,
\]

where $\text{PumpIsOn}$ and $\text{PumpIsOff}$ are defined as $\text{PumpIsOn}(t) \equiv \text{pump}(t) = \text{on}$, and $\text{PumpIsOff}(t) \equiv \text{pump}(t) = \text{off}$.
The conjunction of these three parts forms the complete specification of the control system:

\[ \text{sysCommitment} \equiv \text{sysWaterHigh} \land \text{sysWaterLow} \land \text{sysDanger}. \]

Finally, the top-level specification of the mine pump control system can be written as

\[ \langle \langle \text{true} \rangle \rangle \text{IMinePump}\langle \langle \text{sysCommitment} \rangle \rangle. \]

PVS implementation

**The theory “dSwitch”**

The theory “dSwitch” defines an enumeration type “TSwitch” for the output device register of the mine pump control system. The enumeration consists of two values, \textit{on} and \textit{off}:

```plaintext
dSwitch: THEORY
BEGIN
  TSwitch: TYPE = { on, off };
END dSwitch
```

**The theory “iSystem”**

The theory models the interface of the mine pump control system. The interface (and the modeling theory) has the following parameters:

- Reals \( L, H, M \), which define the low water level, the high water level, and the critical level of methane respectively;

- Time values \( DL, DH, DC \), which define maximum reaction time for the low water, high water, and dangerous situations respectively:

```plaintext
iSystem[
  ( IMPORTING time )
  L: real,      % Low water level
  H: real,      % High water level
  M: real,      % Methane critical level
  DL: Time,     % Deadline for low water
  DH: Time,     % Deadline for high water
  DC: Time     % Deadline for methane critical level
]: THEORY
BEGIN
% Importing datatypes

IMPORTING dSwitch;
```

The interface consists of three device registers — two input device registers of type \textit{RealRegister}, storing real values ( device registers \textit{device} and \textit{methane} ), and the output register of type \textit{SwitchRegister} ( \textit{pump} ), storing a value of type \textit{TSwitch}:

```plaintext
% Input device registers of type real
RealRegister: TYPE = FUNCTION[ Time -> real ];

% Output device registers of type Switch
SwitchRegister: TYPE = FUNCTION[ Time -> TSwitch ];
```
% Water and methane device registers
water : RealRegister;
methane : RealRegister;

% Mine pump control device register
pump: SwitchRegister;

% Variables declaration
t, t1, t2: VAR Time;

The following timed predicates define states of the environment, on which the control system should react — the dangerous, the water low, and the water high situations:

Danger : TimePredicate = LAMBDA t: methane ( t ) >= M;
WaterLow : TimePredicate = LAMBDA t: water ( t ) <= L;
WaterHigh: TimePredicate = LAMBDA t: water ( t ) >= H;

The following timed predicates describe the state of the output device register pump:

PumpIsOff: TimePredicate = LAMBDA t: pump ( t ) = off;
PumpIsOn : TimePredicate = LAMBDA t: pump( t ) = on ;

The predicate sysBehavior is used as a general template for different parts of the top-level specification:

sysBehavior( C, A: TimePredicate, d: Time ): bool =
FORALL t1, t2:
   during( C, cc(t1,t2) ) IMPLIES
   during( A, cc(t1+d,t2) );

The following predicates, sysDanger, sysLow, and sysHigh, describe how the control system should react on different environmental conditions:

sysDanger: bool = sysBehavior( Danger , PumpIsOff, DC );
sysLow : bool = sysBehavior( WaterLow , PumpIsOff, DL );
sysHigh : bool = sysBehavior(
   WaterHigh AND NOT Danger, PumpIsOn , DH
);

The assertion sysCommitment specifies behavior of the control system:

sysCommitment: bool =
sysDanger AND
sysLow AND
sysHigh

END iSystem
4.2. First step simplification

At first, let us make life easier and take non-interesting parts of the system behavior out of the scope. For example, we can get rid of situations when one of the conditions $\text{WaterHigh}$, $\text{WaterLow}$, and $\text{Danger}$ holds $\text{true}$ for a period less than the deadline for the according action.

These thoughts lead to the following specification:

$$sysSimplified(C, A, d) \equiv \forall t_1, t_2 : t_2 \geq t_1 + d \land C \text{ during } [t_1, t_2] \rightarrow A \text{ during } [t_1 + d, t_2].$$

This is a general form of the typical condition/action construction of the top-level specification. It is parameterized by a condition $C$ and an action $A$ (both are time predicates: $C, A \in P$) and a deadline $d$ associated with them. By adding an inequality in the formula, we can get rid of situations where $C$ holds $\text{true}$ for a period shorter than $d$.

The original top-level specification of the control system can be rewritten as follows:

$$sysDangerSimplified \equiv sysSimplified(Danger, PumpIsOff, DC);$$
$$sysLowSimplified \equiv sysSimplified(WaterLow, PumpIsOff, DL);$$
$$sysHighSimplified \equiv sysSimplified(WaterHigh \land \neg Danger, PumpIsOn, DH).$$

Finally, the commitment of the system after this step is a conjunction of the mentioned assertions:

$$sysSimplified \equiv$$
$$sysDangerSimplification \land sysLowSimplification \land sysHighSimplification$$

Using the rule for strengthening the postcondition, one can prove correctness of the simplification step. Using the rule requires the following theorem to be proven:

Theorem S0 (Correctness of simplification)

$$sysSimplified \rightarrow sysCommitment$$

□

In order to prove the theorem, we should prove the following lemmas.

Lemma S1 (Simplification)

$$sysSimplified(C, A, d) \Rightarrow sysBehavior(C, A, d)$$

**Proof outline:**

The validity of the lemma is equivalent to the validity of assertion

$$(\forall t_1, t_2 : t_2 \geq t_1 + d \land C \text{ during } [t_1, t_2] \rightarrow A \text{ during } [t_1 + d, t_2]) \rightarrow$$
$$\forall t_1, t_2 : C \text{ during } [t_1, t_2] \rightarrow A \text{ during } [t_1 + d, t_2]$$

(predicates $sysSimplified$ and $sysBehavior$ are expanded), which validity is equivalent to validity of the following assertion:

$$(t_2 \geq t_1 + d \land C \text{ during } [t_1, t_2] \rightarrow A \text{ during } [t_1 + d, t_2]) \rightarrow$$
$$C \text{ during } [t_1, t_2] \rightarrow A \text{ during } [t_1 + d, t_2],$$

where $t_1$ and $t_2$ are Skolem constants.
Finally, the validity of the last assertion is equal to validity of assertion
\[
(t_z \geq t_i + d \land A \text{ during } [t_i + d, t_z]) \rightarrow A \text{ during } [t_i + d, t_z],
\]
which is valid.

The following three lemmas are particular cases of S1, which can be proven using S1:

**Lemma S2 (Danger Simplification)**

\[
sysDangerSimplified \Rightarrow sysDanger
\]

**Lemma S3 (Low Simplification)**

\[
sysLowSimplified \Rightarrow sysWaterLow
\]

**Lemma S4 (High Simplification)**

\[
sysHighSimplified \Rightarrow sysWaterHigh
\]

Since the lemmas S2, S3, S4 are proven, the proof of the theorem S0 is an obvious one.

The PVS implementation of the simplification step is presented in the next section.

### 4.3. Second step: discretization

The top-level specification given in the previous section describes the output of the control system in a continuous manner by means of device registers. This specification is not convenient if we want to design a discrete program controlling the mine pump. Therefore, the first design step is a transformation into an event-based discrete specification.

This discretization can be done by specifying how to write to the output device register rather than to describe its state. For example, if the specification \textit{PumpsOff during }\[t_i, t_z\] is a subject for discretization then the system should write the value \textit{off} to the device register \textit{pump} before the moment \(t_i\) and should not write any other value before \(t_z\). This idea is used in the following specification pattern:

\[
sysReaction(P \in P, x \in D, d \in \text{TIME}) \equiv \\
\forall t_i, t_z : t_z \geq t_i + d \land (P \text{ during } [t_i, t_z]) \rightarrow \exists t_u : t_u < t_i + d \land \text{write}(pump, x)(t_u) \land \\
\forall y : y \neq x \rightarrow (\neg\text{write}(pump, y)) \text{ during } (t_u, t_z).
\]

This predicate describes a typical reaction on the condition \(P\) : the system writes the value \(x\) to the output device register \textit{pump} before the occurrence of the deadline, denoted by \(t\). It also forbids writing a value different from \(x\) to the device register if \(x\) is already written and the condition \(P\) still holds. We can rewrite the top-level specification using the \textit{sysReaction} predicate:

\[
sysDangerReaction \equiv sysReaction(Danger, off, DC), \\
sysLowReaction \equiv sysReaction(Low, off, DL), \\
sysHighReaction \equiv sysReaction(High \land \neg\text{Danger, on, DH}).
\]
Finally, the commitment of the system after this step is a conjunction of the mentioned assertions:

\[
sys\text{Discrete} \equiv sys\text{DangerReaction} \land sys\text{LowReaction} \land sys\text{HighReaction}.
\]

Using the rule for strengthening the postcondition, one can prove correctness of the discretization step. Using the rule requires the following theorem to be proven:

**Theorem D0 (Correctness of discretization)**

\[
sys\text{Discrete} \rightarrow sys\text{Simplified}
\]

In order to prove the theorem, we should prove the following lemmas.

**Lemma D1 (Danger Discretization)**

\[
sys\text{DangerReaction} \rightarrow sys\text{DangerSimplified}
\]

**Proof outline:**

Straightforward usage of the lemma DR5 (Discrete Write) proves the lemma.

**Lemma D2 (Low Discretization)**

\[
sys\text{LowReaction} \rightarrow sys\text{LowSimplified}
\]

The proof is similar to D1.

**Lemma D3 (High Discretization)**

\[
sys\text{HighReaction} \rightarrow sys\text{HighSimplified}
\]

The proof is similar to D1.

Since the lemmas D1-D3 are proven, it is easy to see that the theorem D0 is correct.

Finally, after the discretization step, we have the following top-level specification of the mine pump control system:

\[
\{\{\text{true}\}\}\text{IMinePump}\{\{\text{sysDiscrete}\}\}
\]

**PVS implementation**

The theory “idesignSystem” implements the simplification and discretization steps. It has the same parameters as the theory “iSystem”, which models the interface of the mine pump control system:

```plaintext
idesignSystem[
    ( IMPORTING time )
    L: real, % Low water level
    H: real, % High water level
    M: real, % Methane critical level
    DL: Time, % Deadline for low water
    DH: Time, % Deadline for high water
    DC: Time % Deadline for methane critical level
]: THEORY
```
BEGIN

% Importing target interface

IMPORTING iSystem[ L, H, M, DL, DH, DC ];

In this theory, the lemma "drDiscreteWrite" is used in order to prove discretization lemmas. Therefore, the device registers semantics has to be imported:

IMPORTING deviceRegister[ TSswitch ];

% Variables declaration

x, y: VAR TSswitch;
d, t1, t2, tw: VAR Time;
C, A: VAR TimePredicate;

Defining the generalized predicate \textit{sysSimplified} and building simplified specification parts from it:

sysSimplified( C, A, d ): bool =
    FORALL t1, t2: t2>=t1+d AND
    during( C, cc(t1,t2) ) IMPLIES
    during( A, cc(t1+d,t2) );

sysDangerSimplified: bool =
    sysSimplified( Danger , PumpIsOff, DC );

sysLowSimplified : bool =
    sysSimplified( WaterLow , PumpIsOff, DL );

sysHighSimplified : bool =
    sysSimplified( WaterHigh AND NOT Danger, PumpIsOn, DH );

The lemmas S1, S2, S3, S4:

Simplification: LEMMA
    sysSimplified( C, A, d ) IMPLIES
    sysBehavior( C, A, d );
% Proven

DangerSimplification: LEMMA
    sysDangerSimplified => sysDanger;
% Proved using "Simplification"

LowSimplification: LEMMA
    sysLowSimplified => sysLow;
% Proven using "Simplification"

HighSimplification: LEMMA
    sysHighSimplified => sysHigh;
% Proven using "Simplification"

After the simplification step, we should make the specification discrete. In order to do that, the generalized predicate "sysReaction" is defined and used to construct parts of the discrete specification:

sysReaction( C, x, d ): bool =
    FORALL t1, t2: t2>=t1+d AND
    during( C, cc(t1,t2) ) IMPLIES
    EXISTS tw: tw<t1+d AND
    write( pump, x ) ( tw ) AND
    FORALL y: y /= x => during( NOT write(pump,y), oc(tw,t2) );
sysDangerReaction: bool = 
sysReaction( Danger , off, DC );
sysLowReaction : bool = 
sysReaction( WaterLow , off, DL );
sysHighReaction : bool = 
sysReaction( WaterHigh AND NOT Danger, on , DH );

The following lemmas are D1, D2, and D3 respectively:

DangerDiscretization: LEMMA
   sysDangerReaction => sysDangerSimplified;
% Proven using "drDiscreteWrite"

LowDiscretization: LEMMA
   sysLowReaction => sysLowSimplified;
% Proven using "drDiscreteWrite"

HighDiscretization: LEMMA
   sysHighReaction => sysHighSimplified;
% Proven using "drDiscreteWrite"

The new commitment of the system is a conjunction of the three specification parts:

   sysDiscrete: bool =
      sysDangerReaction AND
      sysLowReaction AND
      sysHighReaction;

The theorems S0 and D0 are specified in PVS as one:

MainTheorem: THEOREM
   sysDiscrete IMPLIES
      sysCommitment;
% Proven using
% "DangerSimplification", "LowSimplification", "HighSimplification",
% "DangerDiscretization", "LowDiscretization", "HighDiscretization"

END idesignSystem

4.4. Third step: decomposition

This section describes the decomposition of the mine pump control system and the formal specifications of its components.

4.4.1. Decomposition overview

The decomposition of the system is driven by the control flow in the first place and the end-to-end timing constraints in the second place. As a result, the control system consists of four communicating objects as shown in Figure 4. This design is similar to the solution of the slightly more complex mine pump problem from [HvR00b].

Informally, the system works as follows. The sensor objects ("waterSensor" and "methaneSensor") read the device registers for the current water and methane levels. If one of the environmental conditions formulated above is changed, a sensor generates an appropriate message, containing information about the current state of the environment and the deadline for switching the pump on or off. The control object ("control") consumes messages from the sensor objects and decides whether the pump should be switched on or off. If necessary, it sends a control message to the pump control object ("pumpControl") together with the deadline information. The pump control
object reacts on accepted control messages by writing \textit{on} or \textit{off} values to the \texttt{pump} device register before the deadline received with the control message expires.

\begin{center}
\begin{tikzpicture}
  \node (p) [circle, draw] {pump};
  \node (ctl) [circle, draw, below of=p] {ctl};
  \node (status) [circle, draw, below of=ctl] {status};
  \node (control) [circle, draw, left of=status] {control};
  \node (notify) [circle, draw, above of=control] {notify};
  \node (methaneSensor) [circle, draw, below of=notify] {methaneSensor};
  \node (waterSensor) [circle, draw, above of=methaneSensor] {waterSensor};
  \node (water) [circle, draw, left of=waterSensor] {water};
  \node (methane) [circle, draw, left of=methaneSensor] {methane};
  \draw (p) -- (ctl);
  \draw (ctl) -- (status);
  \draw (status) -- (control);
  \draw (control) -- (notify);
  \draw (notify) -- (waterSensor);
  \draw (waterSensor) -- (water);
  \draw (waterSensor) -- (methaneSensor);
  \draw (methaneSensor) -- (methane);
\end{tikzpicture}
\end{center}

\textbf{Figure 4. Decomposition of the system}

\textbf{PVS implementation issues}

Here we describe ports, which are used by all the components of the decomposed system.

\textbf{The theory “pTimed”}

Since all the messages in the system have the same structure (some data and a time value associated with it), we introduce a type for messages, which can transfer a data field and timing information, and a type of ports dealing with them in a separate theory, “TData”, a non-empty type of the data carried with the message parameterizes this theory “pTimed”, which is called “pTimed”:

\begin{verbatim}
pTimed[
    TData: TYPE+
]: THEORY

BEGIN
  IMPORTING time;

The message of type “TTimedMessage” is modeled by the structure consisting of a data field “data” and a field of type “Time”:

  % Timestamped message datatype
  TTimedMessage: TYPE =[#
    data: TData,
    time: Time
  #];

The function “tmsg” constructs the message from a data value and a time value:

  % Message constructor
  tmsg( d: TData, t: Time ): TTimedMessage = (# data:=d, time:=t #);

The lemma “tmType” is auxiliary. It states that each message of type “TTimedMessage” is constructed from its own fields. This lemma makes PVS proofs easier, which use the notion of “TTimedMessage”:

  % Extensionality lemma
  tmType: LEMMA
    FORALL ( x: TTimedMessage ):
      x = (# data:=data(x), time:=time(x) #);
\end{verbatim}
The following code allows us to use the semantics of message passing for messages of type “TTimedMessage” just by importing the theory “pTimed”:

```plaintext
% Importing port theory
IMPORTING port[ TTimedMessage ];

END pTimed
```

**The theory “pWater”**

This theory uses the theory “pTimed” in order to define a type of messages that are sent by the water sensor.

```plaintext
pWater: THEORY
BEGIN

“TWater” is an enumeration, which includes values “high” and “low”. These values describe situations where the water level is high or low respectively.

```plaintext
TWater: TYPE = { high, low };
```

The theory inherits all the definitions specified in the theory “pTimed”. The enumeration “TWater” is used as a type for the data field of messages.

```plaintext
% Constructing timed water messages
IMPORTING pTimed[ TWater ];
```

The new port type is called “portWater”:

```plaintext
% Water port definition
portWater: TYPE = Port;

END pWater
```

**The theory “pMethane”**

This theory uses the theory “pTimed” in order to define a type of messages that are sent by the methane sensor. The theory is similar to “pWater” but uses a different enumeration type for the data field of messages.

```plaintext
pMethane: THEORY
BEGIN

TMethane: TYPE = { normal, danger };

% Constructing timed methane messages
IMPORTING pTimed[ TMethane ];

% Methane port definition
portMethane: TYPE = Port;

END pMethane
```
The theory “pSwitch”

This theory uses the theory “pTimed” in order to define a type of messages that are sent by the control object. The theory is similar to “pWater” and “pMethane”.

\[
\text{pSwitch: THEORY}
\]

\[
\text{BEGIN}
\]

\[
\text{% Importing datatype definition}
\text{IMPORTING dSwitch;}
\]

\[
\text{% Constructing timed switch messages}
\text{IMPORTING pTimed[ TSwitch ];}
\]

\[
\text{% Port definition}
\text{portSwitch: TYPE = Port;}
\]

\[
\text{END pSwitch}
\]

4.4.2. Modeling the water sensor

The water sensor object (“waterSensor”) reads the water device register and generates an appropriate message if a low water level or a high water level situation occurs. The reading of the water level value should be performed sufficiently often to meet both deadlines (\(DL\) and \(DH\)). For the sake of efficiency, the water sensor should not produce two successive messages of the same type (“low” or “high”). The behavior of the water sensor can be formalized as follows, using the function \(tmsg(x,t_x)(t_n)\) to construct a message, containing a data field \(x\) (in this case, it can be either “low” or “high”), a measurement time \(t_x\) and a deadline \(t_n\):

\[
\text{waterReaction}(P \in P, x \in D, t \in \text{TIME}) \equiv \forall t_1, t_2 : t_2 \geq t_1 + t \land (P \text{ during } [t_1, t_2]) \rightarrow \exists t_1, t_n : t_1 < t_1 + t \land \text{notify}(tmsg(x,t_x))(t_n) \land (\neg \text{notify} \text{ during } (t_n, t_2)).
\]

\[
\text{waterLowReaction} \equiv \text{waterReaction}(\text{Low}, \text{low}, DL),
\text{waterHighReaction} \equiv \text{waterReaction}(\text{High}, \text{high}, DH).
\]

The commitment of the water sensor is a conjunction of the two latter assertions:

\[
\text{waterCommitment} \equiv \text{waterLowReaction} \land \text{waterHighReaction}.
\]

Finally, the specification of the water sensor object is

\[
\{\{\text{true}\}\}\text{waterSensor}\{\{\text{waterCommitment}\}\}.
\]
PVS implementation

Theory “iWaterSensor” defines the specification of the water sensor object. The object (and the theory) has four parameters: two real constants, which specify low and high water levels, and two constants of type “Time”, which specify deadlines for low and high water situations:

```pvs
iWaterSensor[
  ( IMPORTING time )
  L: real, % Low water level
  H: real, % High water level
  DL: Time, % Low water deadline
  DH: Time % High water deadline
]: THEORY
```

BEGIN

The water sensor uses a port of type “waterPort”, which is defined in the theory "pWater".

```pvs
% Importing datatypes
IMPORTING pWater;
```

The interface of the water sensor object consists of a real device register, called “waterLevel”, and a port of type “portWater”, which is called “notify”:

```pvs
% Interface elements
% Input device registers of type real
RealRegister: TYPE = FUNCTION[ Time -> real ];
% Water level device register
waterLevel: RealRegister;
% Output port for "high" and "low" messages
notify: portWater;
% Variables declaration
x, y: VAR TWater;
t, tx, ty, tl, t2, tn, tx1, tx2: VAR Time;

% States of the environment
WaterLow : TimePredicate = LAMBDA t: waterLevel( t ) <= L;
WaterHigh: TimePredicate = LAMBDA t: waterLevel( t ) >= H;
```

The water sensor behavior specification is defined as described in the previous section:

```pvs
% Behavior specification

notify: TimePredicate = LAMBDA t:
  EXISTS x, tx: notify( tmsg(x,tx) ) ( t );

waterReaction( P: TimePredicate, x: TWater, d: Time ): bool =
  FORALL t1, t2: t2>=t1+d AND
  during( P, cc(t1,t2) ) IMPLIES
  EXISTS tx,t: tx<t1+d AND
  notify( tmsg(x,tx) ) ( t ) AND
  during( NOT notify, cc(t,t2) );

waterLowReaction : bool = waterReaction( WaterLow , low , DL );
```
4.4.3. Modeling the methane sensor

The behavior of the methane sensor object ("methaneSensor") is similar to the behavior of the water sensor object. The methane sensor reads the methane device register and generates a message reflecting the "current" status of the methane level. The reading of the methane level value should be performed frequently enough to meet both deadlines (DH and DC). For the sake of efficiency, the methane sensor should only produce a message when the situation changes between normal and dangerous methane concentration (message types "normal" or "danger"). The behavior of the methane sensor can be formalized as follows, using the function $\text{msg}$ to construct a message, containing a data field (in this case, it can be either "normal" or "danger") and a deadline:

$$
\text{methaneReaction}(P \in P, x \in D, d \in \text{TIme}) \equiv \forall t_1, t_2 : t_2 \geq t_1 + d \land (P \text{ during } [t_1, t_2]) \Rightarrow \\
\exists t_3, t_4 : t_3 < t_1 + d \land \text{status}(\text{msg}(x, t_3))(t_4) \land ((\neg \text{status}) \text{ during } (t_3, t_4)).
$$

$$
\text{methaneNormalReaction} \equiv \text{methaneReaction}(\neg \text{Danger}, \text{normal}, DH), \\
\text{methaneDangerReaction} \equiv \text{methaneReaction}(\text{Danger}, \text{danger}, DC).
$$

The commitment of the methane sensor is a conjunction of two latter assertions:

$$
\text{methaneCommitment} \equiv \text{methaneNormalReaction} \land \text{methaneDangerReaction}.
$$

Finally, specification of the methane sensor object is

$$
\{\{\text{true}\}\} \text{methaneSensor}\{\{\text{methaneCommitment}\}\}.
$$

PVS implementation

The theory "iMethaneSensor" defines the specification of the methane sensor object. The object (and the theory) has three parameters: a real constant, which specify the dangerous methane level, and two constants of type "Time", which specify deadlines for actions in cases of normal and dangerous methane concentration:

```pascal
iMethaneSensor[
  ( IMPORTING time )
  M: real, % Methane critical level
  DH: Time, % Methane ok deadline
  DC: Time % Methane critical deadline
]: THEORY

BEGIN

The methane sensor uses port of type "methanePort", which is defined in the theory "pMethane".

IMPORTING pMethane;
```
The interface of the methane sensor object consists of an input real device register, called “methaneLevel”, and an output port of type “portMethane”, which is called “status”:

\[
\text{% Input device registers of type real}
\text{RealRegister: TYPE = FUNCTION[ Time -> real ];}
\]

\[
\text{% Methane level device register}
\text{methaneLevel: RealRegister;}
\]

\[
\text{% Output port for "normal" and "danger" messages}
\text{status: Port[TTimedMessage[TMethane]];}
\]

\[
\text{% Variables declaration}
\text{x, y: VAR TMethane;}
\text{t, tx, ty, t1, t2, tn, tx1, tx2: VAR Time;}
\]

\[
\text{% States of the environment}
\text{Danger: TimePredicate = LAMBDA t: methaneLevel( t ) >= M;}
\]

The methane sensor behavior is specified as described above:

\[
\text{% Behavior specification}
\text{status: TimePredicate = LAMBDA t:}
\text{EXISTS x, tx: status( tmsg(x,tx) ) ( t );}
\text{methaneReaction( P: TimePredicate, x: TMethane, d: Time ): bool =}
\text{FORALL t1, t2: t2>=t1+d AND}
\text{during( P, cc(t1,t2) ) IMPLIES}
\text{EXISTS tx,t: tx<t1+d AND}
\text{status( tmsg(x,tx) ) ( t ) AND}
\text{during( NOT status, cc(t,t2) );}
\text{methaneNormalReaction: bool =}
\text{methaneReaction( NOT Danger, normal, DH );}
\text{methaneDangerReaction: bool =}
\text{methaneReaction( Danger , danger, DC );}
\text{methaneCommitment: bool =}
\text{methaneDangerReaction AND}
\text{methaneNormalReaction;}
\text{END iMethaneSensor}
\]

### 4.4.4. Modeling the pump control object

The pump control object (“pumpControl”) switches the pump on or off by writing an appropriate value to the device register *pump* upon the acceptance of activation or deactivation messages from its input port.

The reaction part specifies how an object should react on its inputs and the reason part specifies necessary conditions on input actions for a particular output action:

\[
pumpReaction \equiv \forall x, t_x, t, t_\alpha : \text{accept}(\text{ctl}, tmsg(x,t_x), t)(t_\alpha) \rightarrow
\exists t_\alpha : t_\alpha \in (t_x,t_\alpha) \land \text{write}(pump,x)(t_\alpha),
\]
\[
pumpReason \equiv \forall x, t_u : \text{write}(\text{pump}, x)(t_u) \rightarrow \exists t_u, t_s, t : t_u \in (t_u, t_s) \land accept(ctl, \text{tmsg}(x, t_s), t)(t_u) \land (\neg \text{write}(\text{pump})) ~\text{during}~ [t_u, t_u].
\]

The parts of the specification stated above use the notion of message acceptance and therefore, the specification should include a part stating that all delivered messages are to be eventually accepted:

\[
pumpAccept \equiv \forall x, t_u, t : ctl(\text{tmsg}(x, t_s), t)(t) \rightarrow \exists t_u : accept(ctl, \text{tmsg}(x, t_s), t)(t_u).
\]

The commitment of the methane sensor is a conjunction of the three assertions:

\[
pumpCommitment \equiv pumpAccept \land pumpReaction \land pumpReason.
\]

Finally, the specification of the pump control object is

\[
\{\langle \text{true} \rangle\} \text{pumpControl}\{\langle \text{pumpCommitment} \rangle\}.
\]

PVS implementation

iPump: THEORY
BEGIN

IMPORTING time;

The pump control object uses port of type “portSwitch”, which is defined in the theory “pSwitch”.

IMPORTING pSwitch;

Importing the definition of “write” action for device registers of type “TSwitch”:

IMPORTING deviceRegister[ TSwitch ];

The interface of the pump control object consists of an input port of type “portSwitch”, called “ctl”, and an output device register storing values of type “TSwitch”, which is called “pumpActuator”:

% Interface elements

% Input port for "on" and "off" messages
ctl: portSwitch;

% Output device registers of type Switch
SwitchRegister: TYPE = FUNCTION[ Time \rightarrow \text{TSwitch} ];

% Mine pump control device register
pumpActuator: SwitchRegister;

% Variables declaration

x, y: VAR TSwitch;
t, tx, ty, tl, t2, tw, ta: VAR Time;
Behavior of the pump control object is specified as described in the previous section:

% Behavior specification

pumpAccept: bool = 
FORALL x, tx, t:
  ctl( tmsg(x,tx) )( t ) IMPLIES 
  EXISTS ta:
    accept(ctl, tmsg(x,tx), t) ( ta );

pumpReaction: bool = 
FORALL x, tx, t, ta:
  accept(ctl, tmsg(x,tx), t) ( ta ) IMPLIES 
  EXISTS tw: member( tw, oo(ta,tx) ) AND 
  write( pumpActuator, x ) ( tw );

pumpReason: bool = 
FORALL x, tw:
  write( pumpActuator, x ) ( tw ) IMPLIES 
  EXISTS ta, tx, t: member( tw, oo(ta,tx) ) AND 
  accept(ctl, tmsg(x,tx), t) ( ta ) AND 
  during( NOT write( pumpActuator), oo(ta,tw) );

pumpCommitment: bool = 
  pumpAccept AND
  pumpReaction AND
  pumpReason;

END iPump

4.4.5. Modeling the control object

4.4.5.1. Overview of the control object’s functionality

The control object decides whether the pump should be switched on or off using the information, given by the sensor objects. In order to describe the behavior of the control object, we give a possible sequential implementation of the control object in the simple programming language used in [HvR00a]. The program is shown in Figure 5.

```
water := low;
methane := danger;
while true do
  select receive[?t](notify,water) do
    if (water=high) and (methane=normal) then
      send[!t](ctl,on)
    else if water=low then
      send[!t](ctl,off) fi fi
    or receive[?t](status,methane) do
      if methane = danger then send[!t](ctl,off)
      else if water=high then
        send[!t](ctl,on) fi fi
  endselect
od
```

Figure 5. A possible sequential implementation of the control object.

This program works as follows. It constantly tries to accept messages from the ports “notify” and “status” by the “receive” statements. If the message from the input ports “notify” or “status” is available, the data, carried with the message, is put to the variables “water” or “methane”, and the deadline information (timing parameter of the message) is put to the variable “t”. Then the program
decides whether the message is needed to send to the port “ctl” or not, and the kind of message it has to send. In cases of the “low” and “danger” messages, it sends the “off” message with the received deadline information. In case of the situation, when to subsequent “high” and “normal” messages are accepted, the program sends the “on” message. The notation “[?]” from [HvR00a] expresses getting a time value from the communication channel while “![t]” expresses sending a time value along the channel.

The control object is an interesting object in the sense that neither its specification nor the presented program seem to have any real-time constraints. The control object passes the deadline information to the pump control object just as ordinary data. This means that if the deadline values are changed, this object can be used without any modification of its specification.

The presented program is similar to the process $P_{pump}$ from [HvR00b]. Although the $P_{pump}$ is a process containing timing constraints and the program presented above is not, both implementations are still very close to each other because the behavior of the presented program is timely constrained by the pump control object. Those constraints almost exactly mimic the timing constraints of the $P_{pump}$ process.

4.4.5.2. The object specification framework

In order to verify the decomposition of the system, we need a formal specification of the control object. The specifying was derived in two steps. First, an object specification framework, introduced in this section, was formulated in terms of events and reactions. Second, the control object was specified, using this object specification framework. The use of the specification framework is covered in the next section.

The framework is parameterized by set $D$ interpreted as a set of data values.

We use $E$ and $A$ as notations for sets of events and actions (responses) respectively.

The timed predicate occurred indicates whether the action or event occurred at a time moment with a data parameter:

$$
\text{occurred} : (E \cup A) \times D \rightarrow P .
$$

The predicate enables holds true if an event needs a response (an action) on it:

$$
\text{enables} : E \times A \rightarrow B .
$$

The predicate ordered holds true if a pair of events is meant to be ordered:

$$
\text{ordered} : E \times E \rightarrow B .
$$

The behavior of the specified object can be described by the following assertions.

The first one, Reaction, states that if an event has been occurred, all the actions enabled by this event have to be occurred after it:

\[
\text{Reaction} \equiv \forall e \in E, a \in A, x \in D, t_e \in \text{TIME} : \\
\text{occurred}(e, x)(t_e) \land \text{enables}(e, a) \rightarrow \exists t_a \in \text{TIME} : \text{occurred}(a, x)(t_a) \land t_e < t_a .
\]
The second, *Reason*, states that if an action has been occurred, it had to have a cause — an event that enables it:

\[
\text{Reason} \equiv \forall a \in A, x \in D, t_e \in \text{TIME}: \text{occurred}(a, x)(t_e) \rightarrow \exists e \in E, t_e \in \text{TIME}:
\text{occurred}(e, x)(t_e) \land \text{enables}(e, a) \land t_e < t_a.
\]

The third assertion, *TimeOrder*, formalizes the relaxed order property, mentioned in the previous section. It states that the order of a pair of actions should be the same as the order of the events caused them if these events are meant to be ordered and these two pairs event-action are distinguishable from each other:

\[
\text{TimeOrder} \equiv \forall e_1, e_2 \in E, a_1, a_2 \in A, x_1, x_2 \in D, t_e_1, t_e_2, t_a_1, t_a_2:\
\text{ordered}(e_1, e_2) \land
\text{enables}(e_1, a_1) \land \text{enables}(e_2, a_2) \land
(x_1 \neq x_2 \lor (e_1 \neq e_2 \land a_1 \neq a_2)) \land
\text{occurred}(e_1, x_1)(t_e_1) \land \text{occurred}(a_1, x_1)(t_a_1) \land t_e_1 < t_a_1 \land
\text{occurred}(e_2, x_2)(t_e_2) \land \text{occurred}(a_2, x_2)(t_a_2) \land t_e_2 < t_a_2 \Rightarrow
\]

\[
t_e_1 < t_e_2 \iff t_a_1 < t_a_2.
\]

Finally, a conjunction of the three assertions defines reactive behavior of an object:

\[
\text{reactiveSemantics} \equiv \text{Reaction} \land \text{Reason} \land \text{TimeOrder}.
\]

**PVS implementation**

The theory "reactive" has seven parameters: data type D (called "Data" here), set of events E ("Events"), set of actions A ("Actions"), two versions of occurred (for events and for actions), and predicates enables and ordered:

```pvs
reactive[
( IMPORTING time )
Data : TYPE,
Events : TYPE,
Actions : TYPE,
occurred : FUNCTION[ Events, Data -> TimePredicate ],
occurred : FUNCTION[ Actions, Data -> TimePredicate ],
enables : FUNCTION[ Events, Actions -> bool ],
ordered : FUNCTION[ Events, Events -> bool ]
]: THEORY
BEGIN

  e, el, e2: VAR Events;
a, al, a2: VAR Actions;
t, ts, ta, te, te1, te2, ta1, ta2: VAR Time;
x, x1, x2: VAR Data;

% Reactive semantics

reactiveReaction: bool =
  FORALL e, a, x, te:
    occurred(e, x)(te) AND
    enables(e, a) IMPLIES
    EXISTS ta:
      occurred(a, x)(ta) AND te<ta;

reactiveReason: bool =
  FORALL a, x, ta:
    occurred(a, x)(ta) IMPLIES
```

41
EXISTS e,te:
    occurred(e,x)(te) AND enables(e,a) AND te<ta;

reactiveOrder: bool =
    FORALL e1, a1, tel, tal, x1, e2, a2, te2, ta2, x2:
        ordered(e1, e2) AND
        enables(e1, a1) AND enables(e2, a2) AND
        (x1=x2 OR (a1=a2 AND e1=e2)) AND
        occurred(e1, x1)(tel) AND occurred(a1, x1)(tal) AND tel<tal AND
        occurred(e2, x2)(te2) AND occurred(a2, x2)(ta2) AND te2<ta2 IMPLIES {
            tel < te2 IFF tal < ta2
        };

reactiveSemantics: bool =
    reactiveReaction AND
    reactiveReason AND
    reactiveOrder;

END reactive

4.4.5.3. Specification of the control object

The behavior of the control object is specified using the object specification framework described in
the previous section. It means that the sets E and A have to be defined as well as the functions
occurred, ordered, and enables. The data set D, the parameter of the specification framework,
is defined to be equal to TIME.

The sets of events and actions are defined as E ≡ {eDanger, eLow, eHighAndSafe} and
A ≡ {aOn, aOff} respectively. Each event and each action is a timed predicate parameterized by
timing value:

\[ \forall z \in E \cup A : (z : D \rightarrow P) . \]

The first event, eDanger, is defined as an acceptance of a message from the methane sensor
notifying that the methane concentration is dangerous. Parameter of this event is deadline
information passed with the message:

\[ eDanger(t_m)(t_a) \equiv \exists t : accept(methane,tmsg(danger,t_m),t)(t_a) . \]

The second event, eLow, is defined as an accepting of a message from the water sensor notifying
that the water level is low. Parameter of the event is the deadline information again:

\[ eLow(t_w)(t_a) \equiv \exists t : accept(water,tmsg(low,t_w),t)(t_a) . \]

The third event, eHighAndSafe, is defined as an unbroken sequence of two messages: the
“normal” message from the methane sensor and the “high” message from the water sensor. It
occurs when both messages (“high” and “normal”) are accepted and the event’s parameter is the
maximum of the two timing values carried by the messages:

\[ eHighAndSafe(t_a) \equiv \exists t_a, t_m, t_w, t_n, ta_m, ta_n, \in TIME : \]
\[ \begin{align*}
    & accept(water,tmsg(high,t_a),t_a)(ta_m) \land \\
    & accept(methane,tmsg(normal,t_n),t_n)(ta_m) \land \\
    & during(\neg accept(water),(ta_m,t_a)) \land \\
    & during(\neg accept(methane),(ta_m,t_n)) \land \\
    & t_a = \max(ta_m,ta_n) \land t = \max(t_a,t_a) .
\end{align*} \]
Actions are defined as sending of messages enabling or disabling the pump:

\[
a_{\text{On}}(t)(t_a) \equiv \text{ctl}(\text{tmsg}('\text{on},t))(t_a),
\]

\[
a_{\text{Off}}(t)(t_a) \equiv \text{ctl}(\text{tmsg}('\text{off},t))(t_a).
\]

If we want to define the function \( \text{occurred} \), we could use the fact that all the events and actions are defined as timed predicates:

\[\text{occurred}(z \in E \cup A, t \in D) \equiv z(t).\]

The predicate \( \text{enables} \) can be defined as follows: only events \( e_{\text{Danger}} \) and \( e_{\text{Low}} \) enable the action \( a_{\text{Off}} \); only \( e_{\text{HighAndSafe}} \) enables the action \( a_{\text{On}} \):

\[
\text{enables}(e, a) \equiv \begin{cases} 
  e = e_{\text{Danger}} & a = a_{\text{Off}} \lor \\
  e = e_{\text{Low}} & a = a_{\text{Off}} \lor \\
  e = e_{\text{HighAndSafe}} & a = a_{\text{On}} 
\end{cases}.
\]

Only events related to the same input port are ordered:

\[
\text{ordered}(e_1, e_2) \equiv \begin{cases} 
  e_1 = e_2 \lor \\
  e_1 = e_{\text{Low}} \land e_2 = e_{\text{HighAndSafe}} \lor \\
  e_1 = e_{\text{Danger}} \land e_2 = e_{\text{HighAndSafe}} \lor \\
  e_1 = e_{\text{HighAndSafe}} \land e_2 = e_{\text{Low}} \lor \\
  e_1 = e_{\text{HighAndSafe}} \land e_2 = e_{\text{Danger}} 
\end{cases}.
\]

Events, as we defined them, use the notion of message acceptance and therefore, the specification should include a part stating that all delivered messages are to be eventually accepted:

\[
\text{ctlAcceptWater} \equiv \forall w, t_w, t: \\
\text{water}(\text{tmsg}(w, t_w))(t) \rightarrow \exists t_a : \text{accept}(\text{water}, \text{tmsg}(w, t_w), t)(t_a),
\]

\[
\text{ctlAcceptMethane} \equiv \forall m, t_m, t: \\
\text{methane}(\text{tmsg}(m, t_m))(t) \rightarrow \exists t_a : \text{accept}(\text{methane}, \text{tmsg}(m, t_m), t)(t_a).
\]

The commitment of the control object is the conjunction of the abstract framework semantics and the two acceptance assertions:

\[
\text{ctlCommitment} \equiv \text{reactiveSemantics} \land \text{ctlAcceptWater} \land \text{ctlAcceptMethane}.
\]

Finally, the specification of the control object looks like:

\[
\{\{\text{true}\}\}\text{control}\{\{\text{ctlCommitment}\}\}.
\]

**PVS implementation**

The theory “iControl” specified the behavior mentioned above in a straightforward way:

\[
i\text{Control}: \text{THEORY}
\]

\[
\text{BEGIN}
\]
IMPORTING time;
%
Importing port types

IMPORTING pWater;
IMPORTING pMethane;
IMPORTING pSwitch;

Interface of the object consists of two input ports “water” and “methane” for messages from the water and methane sensors and an output port “ctl”:

%
Interface elements

%
Input ports
water : portWater;
methane: portMethane;
%
Output port
ctl: portSwitch;

The following part of the theory is a straightforward specification of the control object:

%
Reactive behavior specification

Events : TYPE = { eDanger, eLow, eHighAndSafe };
Actions: TYPE = { aOn, aOff };

e, el, e2: VAR Events;
a, a1, a2: VAR Actions;
t, ta, te: VAR Time;
th, tw, tn, tm, tam: VAR Time;

eDanger( tm ): TimePredicate = LAMBDA ta:
   EXISTS t:
      accept( methane, tmsg(danger,tm), t )( ta );

eLow( tw ): TimePredicate = LAMBDA ta:
   EXISTS t:
      accept( water, tmsg(low,tm), t )( ta );

eHighAndSafe( t ): TimePredicate = LAMBDA ta:
   EXISTS th, tw, tn, tm, tam:
      accept( water, tmsg(high,th), tw )( tw ) AND
      accept( methane, tmsg(normal,tm), tm )( tm ) AND
      during( NOT accept(water), oc(taw,ta) ) AND
      during( NOT accept(methane), oc(tam,ta) ) AND
      ta = max( taw, tam ) AND
      t = max( th , tn );

aOff( t ): TimePredicate = LAMBDA ta: ctl( tmsg(off,t ))( ta );
aOn ( t ): TimePredicate = LAMBDA ta: ctl( tmsg(on ,t ))( ta );

occurred( e, t ): TimePredicate = COND
   e = eDanger    -> eDanger    ( t ),
   e = eLow       -> eLow       ( t ),
   e = eHighAndSafe -> eHighAndSafe( t )
ENDCOND;

occurred( a, t ): TimePredicate = COND
   a = aOn        -> aOn   ( t ),
   a = aOff      -> aOff  ( t )
ENDCOND;

enables( e, a ): bool = COND
\[ e = \text{eDanger} \rightarrow a = \text{aOff}, \]
\[ e = \text{eLow} \rightarrow a = \text{aOff}, \]
\[ e = \text{eHighAndSafe} \rightarrow a = \text{aOn}, \]
ELSE \rightarrow false
ENDCOND;

ordered( e1, e2 ): bool = e1=e2 OR COND
el = eLow \rightarrow e2 = eHighAndSafe,
el = eDanger \rightarrow e2 = eHighAndSafe,
el = eHighAndSafe \rightarrow e2 = eLow OR e2 = eDanger,
ELSE \rightarrow false
ENDCOND;

IMPORTING reactive{
    Time,
    Events, Actions,
    occurred, occurred,
    enables, ordered
};

% Progress properties specification

w: VAR TWater;
m: VAR TMethane;

ctlAcceptWater: bool =
    FORALL w, tw, t:
        water(tmsg(w,tw))(t) => EXISTS ta:
            accept(water,tmsg(w,tw),t)(ta);

ctlAcceptMethane: bool =
    FORALL m, tm, t:
        methane(tmsg(m,tm))(t) => EXISTS ta:
            accept(methane,tmsg(m,tm),t)(ta);

ctlCommitment: bool =
    reactiveSemantics AND
ctlAcceptWater AND
ctlAcceptMethane;

END "iControl"

4.4.6. Correctness analysis of the decomposition

To prove the correctness of the presented decomposition we use the parallel composition rule from [HvR00a] and the formal semantics of message passing.

The parallel composition rule used for this example is a simplified version of the more general rule from [Hoo94]. It is formulated as

\[
\frac{\langle A_1 \rangle P_1 \langle A_2 \rangle P_1 \langle C_1 \rangle \langle C_2 \rangle}{\langle A_1 \land A_2 \rangle P_1 \parallel P_2 \langle C_1 \land C_2 \rangle}
\]

provided that assertions in the specification of an object refer only to the interface of the object itself.

Using the consequence rule from [Hoo94] for strengthening the commitment, the following proposition states the correctness of the decomposition presented in the previous section:
Proposition C0 (Decomposition Correctness)

\[
\text{waterCommitment} \land \text{methaneCommitment} \land \\
\text{pumpCommitment} \land \text{ctlCommitment} \rightarrow \\
\text{sysDiscrete}.
\]

The attempt to prove this proposition in PVS failed because of the incorrect timing behavior of the control object. The underlying problem is discussed in the next section.

PVS implementation

The theory “cSystem” defines an implementation of the system’s interface:

cSystem[
  ( IMPORTING time )
  L: real, % Low water level
  H: real, % High water level
  M: real, % Methane critical level
  DL: Time, % Deadline for low water
  DH: Time, % Deadline for high water
  DC: Time % Deadline for methane critical level
]: THEORY

BEGIN

% Importing the class interface

  system: THEORY = idesignSystem[ L, H, M, DL, DH, DC ];

% Importing the contained objects

  water : THEORY = iWaterSensor[ L, H, DL, DH ];
  methane: THEORY = iMethaneSensor[ M, DH, DC ];
  control: THEORY = iControl;
  pump : THEORY = iPump;

% Shortcuts for the proof

  TWaterMsg : TYPE = TTimedMessage[TWater ];
  TMethaneMsg: TYPE = TTimedMessage[TMethane];
  TSwitchMsg : TYPE = TTimedMessage[TSwitch ];

% Setting up connections

  Connections: AXIOM
    water = waterLevel AND
    methane = methaneLevel AND
    pump = pumpActuator AND
    water.notify = control.water AND
    methane.status = control.methane AND
    control.ctl = pump.ctl;

% Correctness condition: the parallel composition rule

  CorrectCommitment: THEOREM
    waterCommitment AND
    methaneCommitment AND
    ctlCommitment AND
    pumpCommitment IMPLIES
4.5. Temporal inaccuracy

The decomposition of the mine pump control system intuitively seems to be correct, but actually it is not. This incorrectness has a very subtle nature and was discovered during the formal verification of the control system. This fact shows that formal analysis is indeed an important issue for the development of real-time systems.

Figure 6 shows a scenario for which the system reacts in a wrong way.

![Diagram](image)

**Figure 6. An incorrect behavior of the control system.**

In this scenario, we see that the methane sensor object sends a message “danger”, indicating a high concentration of methane, to the control object. The control object reacts message by sending a message “off” to the pump control object. Next, the water sensor object sends a message “low” to the control object. The control object reacts to this message by sending another “off” message. Next, the water sensor object sends a message “high” to the control object, but the acceptance of this message is delayed. Shortly after the sending of the “high” message by the water sensor object, the methane sensor object sends a message “normal” indicating that the methane concentration is not dangerous anymore. As we can see, the control object receives this message, but still did not receive the message “high” from the water sensor object. After a period larger than $DH$, the methane sensor object detects a dangerous methane concentration again and sends a message “danger” to the control object. Notice that the message “high”, originating from the water sensor object, still has not arrived at the control object. Finally, the control object receives this “high” message and reacts with another “off” message. As we can see, although there is a period that lasts longer than $DH$ time units in which the water level is high, and there is no dangerous concentration of methane gas around the pump, the pump is not switched on. Clearly this is not what is stated by the top-level specification of the system.

The problem is caused by the fact that in a purely asynchronous system, the control object is not able to exactly mimic the state of the environment. To do so, it depends on messages it receives from the sensor objects, containing information about the environment. The validity of this information is time-dependent and thus invalidated by the progression of real-time.

The pump control object is the only component that implements the restriction of the temporal behavior of the system (the behavior of sensor objects is also constrained but it is irrelevant for
meeting the deadline for this situation). This temporal restriction is implied by the values of the parameters received with the “on” and “off” messages. Or, in other words, a timing constraint only holds when a proper reaction of the pump control object takes place.

### 4.6. Correcting the control object

#### 4.6.1. The corrected specification of the control object

In order to avoid the “Temporal inaccuracy” problem, additional constraints must be put on the timing behavior of the control object:

\[
\begin{align*}
\text{ctlAcceptWaterRT} & \equiv \forall x,t_x,t_z.t_a : \text{accept(water, tmsg(x,t_z),t)(t_a)} \rightarrow t_a < t_z, \\
\text{ctlAcceptMethaneRT} & \equiv \forall x,t_x,t_z.t_a : \text{accept(methane, tmsg(x,t_z),t)(t_a)} \rightarrow t_a < t_z.
\end{align*}
\]

These constraints set the minimal rate for accepting the messages from the sensors and thus the maximal message delay.

This can be implemented by constraining the “receive” statements of the program. The added code is underlined in the fragment in Figure 7.

```plaintext
water := low;
methane := danger;
while true do
  select receive[?m](wp, water) do [<m]
    if (water=high) and (methane=normal) then
      send[!m](pp, on)
    else if water=low then
      send[!m](pp, off) fi fi
  or receive[?m](mp, methane) do [<m]
    if methane = danger then send[!m](pp, off)
  endselect
  send[!m](pp, on) fi fi
endselect

Figure 7. The constrained sequential implementation of the control object.
```

Informally, an execution moment of the annotated statement should satisfy the requirement specified in the timing annotation. In this program, it is required that both “receive” statements are executed before the deadline specified in the received message. Note that the timing parameters are logical values that specify the scheduling of an implementation but may never appear in the scheduled implementation. Further details about timing annotations and their formal semantics can be found in [HvR00a]. The implementation of a system using timing annotations is demonstrated in [HvR00b].

#### PVS implementation

```plaintext
iControl{
  ( IMPORTING time )
  fDW: FUNCTION[ Time -> Time ],
  fDM: FUNCTION[ Time -> Time ]
  % Functions used to calculate the deadlines, controlling
  % minimal "water" and "methane" messages acceptance rates
}: THEORY

BEGIN
```

48
% Importing port types

IMPORTING pWater;
IMPORTING pMethane;
IMPORTING pSwitch;

% Interface elements

% Input ports
water : portWater;
methane: portMethane;

% Output port
ctl: portSwitch;

% Reactive behavior specification

Events : TYPE = { eDanger, eLow, eHighAndSafe };
Actions: TYPE = { aOn, aOff };
e, e1, e2: VAR Events;
a, a1, a2: VAR Actions;
t, ta, te: VAR Time;
th, tw, taw, tn, tm, tam: VAR Time;

eDanger( tm ) : TimePredicate = LAMBDA ta:
   EXISTS t:
     accept( methane, tmsg(danger,tm), t )( ta );

eLow( tw ) : TimePredicate = LAMBDA ta:
   EXISTS t:
     accept( water, tmsg(low,tw), t )( ta );

eHighAndSafe( t ) : TimePredicate = LAMBDA ta:
   EXISTS th, tw, taw, tn, tm, tam:
     accept( water , tmsg(high ,th), tw )( taw ) AND
     accept( methane, tmsg(normal,tn), tm )( tam ) AND
     during( NOT accept(water ), oc(taw,ta) ) AND
     during( NOT accept(methane), oc(tam,ta) ) AND
     ta = max( taw, tam ) AND
     t = max( th , tn );

aOff( t ) : TimePredicate = LAMBDA ta: ctl( tmsg(off,t ) )( ta );
aOn ( t ) : TimePredicate = LAMBDA ta: ctl( tmsg(on,t ) )( ta );

occurred( e, t ) : TimePredicate = COND
   e = eDanger    -> eDanger ( t ),
   e = eLow       -> eLow  ( t ),
   e = eHighAndSafe -> eHighAndSafe( t )
ENDCOND;

occurred( a, t ) : TimePredicate = COND
   a = aOn        -> aOn ( t ),
   a = aOff -> aOff( t )
ENDCOND;

enables( e, a ) : bool = COND
   e = eDanger    -> a = aOff,
   e = eLow       -> a = aOff,
   e = eHighAndSafe -> a = aOn,
   ELSE             -> false
ENDCOND;
ordered( e1, e2 ): bool = e1=e2 OR COND
  e1 = eLow -> e2 = eHighAndSafe,
  e1 = eDanger -> e2 = eHighAndSafe,
  e1 = eHighAndSafe -> e2 = eLow OR e2 = eDanger,
ELSE -> false
ENDCOND;

IMPORTING reactive{
  Time,
  Events, Actions,
  occurred, occurred,
  enables, ordered
};

% Progress properties specification

w: VAR TWater;
m: VAR TMethane;

ctlAcceptWater: bool = 
  FORALL w, tw, t:
    water(tmsg(w,tw))(t) => EXISTS ta: accept(water,tmsg(w,tw),t)(ta);

ctlAcceptMethane: bool = 
  FORALL m, tm, t:
    methane(tmsg(m,tm))(t) => EXISTS ta: accept(methane,tmsg(m,tm),t)(ta);

% Real-time properties specification

ctlAcceptWaterRT: bool = 
  FORALL w, t, tw, taw:
    accept(water,tmsg(w,t),tw)(taw) => taw<fDN(t);

ctlAcceptMethaneRT: bool = 
  FORALL m, t, tm, tam:
    accept(methane,tmsg(m,t),tm)(tam) => tam<fDM(t);

ctlCommitment: bool = 
  reactiveSemantics AND
  ctlAcceptWater AND
  ctlAcceptMethane AND
  ctlAcceptWaterRT AND
  ctlAcceptMethaneRT;

END iControl
4.6.2. Correctness analysis of the corrected decomposition

Similar to the proposition C0 (Decomposition Correctness) from section 3.4.6, the correctness condition for the decomposition of the system with a corrected control object can be formulated by the following theorem:

Theorem C1 (New Decomposition Correctness)
The combination of the sensors, the control object, and the pump control object must implement the mine pump control system:

\[
\text{waterCommitment} \land \text{methaneCommitment} \land \text{ctlCommitment} \land \text{pumpCommitment} \implies \text{sysDiscrete}.
\]

In order to prove this theorem, we formulate lemmas, which reflect three relevant scenarios for the system.

First, we assess the danger methane situation:

Lemma C2 (Danger Reaction)
The system must switch the pump off correctly if the methane level has become dangerous:

\[
\text{methaneDangerReaction} \land \text{ctlAcceptMethane} \land \text{reactiveSemantics} \land \\
\text{pumpAccept} \land \text{pumpReaction} \land \text{pumpReason} \implies \text{sysDangerReaction}.
\]

Proof outline:
In order to show validity of this lemma, let's consider the scenario of showed in Figure 7.

![Diagram showing methane level measurement and control actions](Image)

Figure 7. A scenario with dangerous level of methane.

In order to prove the lemma, we have to prove, that

1) The value “off” is written to the “pump” device register before the deadline;

2) No “on” value is written to “pump” during the dangerous interval after the pump is switched off.
The first point can be proven by straightforward expansion of the assertions `methaneDangerReaction`, `ctlAcceptMethane`, `reactiveSemantics`, `pumpAccept`, and `pumpReaction`.

In order to prove the second point, assume that the “on” value is written to the device register “pump” during the dangerous interval after the value “off” is written. Using `pumpReason`, we can say that the message “on” preceded this write to the device register but this message was accepted after the message “off” was accepted. Using `reactiveSemantics`, we can say that the message “normal” was sent before the “off” message but after the message “danger”. But according to `methaneDangerReaction`, this “normal” message could not be sent while the methane level is dangerous. Therefore, the value “on” cannot be written to “pump” by the control system after “off” message while the methane level is dangerous. This concludes validity of the lemma.

The second lemma reflects the scenario with low water level:

**Lemma C3 (Water Low Reaction)**

The system must switch the pump off correctly if the water level has become low:

\[
\text{waterLowReaction} \land \text{ctlAcceptWater} \land \text{reactiveSemantics} \land \text{pumpAccept} \land \text{pumpReaction} \land \text{pumpReason} \Rightarrow \text{sysLowReaction}.
\]

This lemma can be proven similarly to the lemma C2 (Danger Reaction).

Finally, the following lemma ensures the correct behavior of the system in case the water level is high and the level of methane is safe:

**Lemma C4 (Water High Reaction)**

A system must switch the pump on correctly if the water level has become high and the methane level is non-dangerous:

\[
\text{waterHighReaction} \land \text{methaneNormalReaction} \land \\
\text{ctlAcceptWater} \land \text{ctlAcceptMethane} \land \text{reactiveSemantics} \land \\
\text{ctlAcceptWaterRT} \land \text{ctlAcceptMethaneRT} \land \\
\text{pumpAccept} \land \text{pumpReaction} \land \text{pumpReason} \Rightarrow \text{sysHighReaction}.
\]

The proof of this lemma is similar to the proof of C1 (Danger Reaction). The major difference between the proofs is that we have to ensure that the “temporal inaccuracy” problem does not occur in the system. As we informally showed in section 3.5, the constraints `ctlAcceptWaterRT` and `ctlAcceptMethaneRT` solves the problem.

It is easy to see that the validity of the lemmas C2-C4 leads to the validity of the theorem C1 (Decomposition Correctness). Having this theorem proven, we can state that the given decomposition of the mine pump control system with the improved specification of the control object is indeed correct.
5. Final remarks and conclusions

5.1. Lessons learned

In this section, we discuss the main results derived from the mine pump case study: the design philosophy used in the project is discussed in section 5.1.1, a three-tier architecture for embedded systems and a reusable generic sensor object are presented in sections 5.1.2 and 5.1.3, sections 5.1.4 and 5.1.5 contain discussions on two specification approaches used for the components of the mine pump control system, and the future extensions of our generic object specification framework are discussed in section 5.1.6.

5.1.1. Postponing design decisions

The design philosophy used in this project is to postpone design decisions as much as possible. The following list contains design aspects, where this philosophy succeeded:

- The specification of the system in terms of end-to-end timing constraints allows the postponing of the scheduling activity to the second phase of the development process. This greatly simplifies the formal specification and verification of the system.
- Device registers allow the specification of systems with a continuously changing environment without giving implementation details of the polling mechanism.
- The relaxed time order of events as described in section 5.1.5.

5.1.2. Three-tier architecture

The 3-tier architecture is often used in client-server applications. Such an application consists of three layers, or tiers, which are called “user interface layer”, “application layer”, and “database layer”.

The advantage of such a layer structure is the strict separation of concerns, which results in functionally independent layers and clear layer interfaces. As long as the interfaces are not changed, the functionality and implementation of one layer can be changed without consequences for the other layers.

Similar to a client-server architecture, also the architecture of a reactive system can be split into three independent layers with clearly defined interfaces: the sensor layer that processes the input, the control layer that performs the control logic and the actuator layer that processes the output. This principle was used for the architecture of the mine pump control system. The control system of the mine pump consists of four objects (Figure 8).

The three layers have the following functionality:

- Sensor layer: Sensors observe the environment and send messages to the control layer if the environment is changed. The sensor specification may involve timing constraints, which follow from the top-level specification, but in general, their functionality is not directly related to the top-level specification of the system. If the specification of the sensor objects has timing constraints, these are passed with the appropriate messages to the actuator layer through the control layer.
- Control layer: Objects from this layer provide the logical control of the system. Since the control objects pass timing information coming from the sensors to the actuator, the objects from this layer have specifications, which include as little real-time information as possible.
- Actuator layer: The actuator objects are responsible for the output of the system to the environment and for the enforcement of the timing constraints. The latter are passed by
messages from the sensor layer through the control layer. They are enforced by means of a proper schedule. Although this layer can be very thin, it is very important because it makes the control part of the system as non-real-time as possible.

![Diagram](image)

**Figure 8. Three layers of the mine pump control system.**

One benefit of this 3-tier architecture for embedded systems is that the control logic can be used in systems with different inputs and outputs. For example, if the environment of the mine pump control system changes from continuous, to discrete (for example, interrupts can be generated if the water level exceeds the limits), the control part of the system remains the same and the sensors are the only objects that change.

The other benefit of using this architecture for real-time control systems is the fact that the specification of the control logic objects does (in the ideal case) not depend on the timing specification of the system. Thus, the control layer can be used within different real-time (or even non-real-time) systems with various timing constraints.

### 5.1.3. Reusable generic sensor object

After the design of the mine pump control system, we realized that the sensor object design can be generalized by using the “reaction/reason” specification pattern at one hand, and the common sensor functionality at the other hand.

We use the following model for the functionality of the sensor object. The sensor (Figure 9) eventually reads the input device register “input” and sends notification messages to the output port “notification” if the value stored in “input” satisfies a certain condition $P$. The moment of reading the device register is recorded and can be used later (in other parts of the system) to specify deadlines.

This informal description can be formalized as follows, using the function $\text{tmsg}(x, t_x)$ to construct a message, containing a data field $x$ and a deadline $t_x$:

$$
\text{sensorReaction}(P \in P, v \in D, d \in \text{TIME}) \equiv \forall t_1, t_2 \in \text{TIME} : t_2 \geq t_1 + d \land $$

$$P \text{ during } [t_1, t_2] \rightarrow \exists t_x, t_n \in \text{TIME} : t_x < t_1 + d \land \text{notify(tmsg}(x, t_x))(t_n).$$

The “reason” part of the specification is needed to prevent generation of the erroneous notification messages:

$$
\text{sensorReason}(P \in P, v \in D, d \in \text{TIME}) \equiv \forall t_x, t_n \in \text{TIME} :$$

$$\text{notify(tmsg}(x, t_x))(t_n) \rightarrow \exists t \in \text{TIME} : t_x < t + d \land P(t).$$
The commitment for a sensor is a conjunction of the “reaction” and “reason” assertions, parameterized by different conditions for a message generation \( (P) \), data values \( (x) \), and deadlines \( (d) \).

![Diagram of Sensor Interface]

Figure 9. The interface of the sensor object.

In the mine pump example, we used a slightly different type of assertions:

\[
\begin{align*}
\text{sensorBehavior}(P \in P, v \in D, d \in \text{TIME}) & \equiv \\
\forall t_1, t_2 \in \text{TIME} : t_2 \geq t_1 + d \land P \text{ during } [t_1, t_2] \rightarrow \\
\exists t_n \in \text{TIME} : t_n < t_1 + d \land \\
& \text{notify}(\text{msg}(x, t_n))(t_n) \land \neg \text{notify during } (t_n, t_2).
\end{align*}
\]

This specification is somewhat more restrictive than the previous one because it forbids subsequent notifications. On the other hand, \text{sensorBehavior} alone does not imply \text{sensorReason} because, according to \text{sensorBehavior}, a notification message may be generated without good reason. If the designer wants to prevent that, the sensor specification must imply a conjunction of \text{sensorBehavior} predicates, provided that the arguments \( P \) cover all possible states of the environment.

### 5.1.4. Interval-based specification style

During the mine pump case study several specification techniques were attempted to describe the mine pump control system. One of them is called “interval-based specification style”.

The top-level specification of the mine pump control system is presented in section 4.1 of this report. This specification is a conjunction of three assertions:

\[
\begin{align*}
\text{sysCommitment} & \equiv \\
& \forall t_1, t_2 : (\text{WaterHigh} \land \neg \text{Danger}) \text{ during } [t_1, t_2] \rightarrow \\
& (\text{pump = off}) \text{ during } [t_1 + DH, t_2] \land \\
& \forall t_1, t_2 : \text{WaterLow during } [t_1, t_2] \rightarrow (\text{pump = off}) \text{ during } [t_1 + DL, t_2] \land \\
& \forall t_1, t_2 : \text{Danger during } [t_1, t_2] \rightarrow (\text{pump = off}) \text{ during } [t_1 + DC, t_2].
\end{align*}
\]

The naive implementation of this control system is described in [vdLaan00].

The water sensor measures the water level and sends the message “low” if the water level is low and the message “high” if the water level is high. Each of those messages contains two timestamps: the deadline for the switching of the pump (represented by \( d \)) and a time interval, which length is less than the distance between the latest deadline (namely, \( t_1 + DL \) or \( t_1 + DH \)) and the actual deadline (\( d \)). Sending of a message “low” with timing parameters \( d \) and \( \Delta \) is denoted by the predicate \( \text{low}(d, \Delta) \) in the following specification. The assertion \( \neg \text{low}(t_1, t_2) \)
holds “true” if the water sensor never send the “low” message with the first timing parameter from the interval \([t_1, t_2]\). The same notation is used for the “high” messages:

\[
\text{Water} \equiv \forall t_1, t_2 : t_2 \geq t_1 + DL \land \text{WaterLow during } [t_1, t_2] \rightarrow \\
\exists d, \Delta : d \leq t_1 + DL \land \Delta \leq t_1 + DL - d \land \text{low}(d, \Delta) \land \neg \text{high}(d, t_2) \land \\
\forall t_1, t_2 : t_2 \geq t_1 + DH \land \text{WaterHigh during } [t_1, t_2] \rightarrow \\
\exists d, \Delta : d \leq t_1 + DH \land \Delta \leq t_1 + DH - d \land \text{high}(d, \Delta) \land \neg \text{low}(d, t_2).
\]

The methane sensor works similarly to the water sensor. We use the same notation for “danger” and “normal” messages as for “low” and “high” messages.

\[
\text{Methane} \equiv \forall t_1, t_2 : t_2 \geq t_1 + DC \land \text{Danger during } [t_1, t_2] \rightarrow \\
\exists d, \Delta : d \leq t_1 + DC \land \Delta \leq t_1 + DC - d \land \text{danger}(d, \Delta) \land \neg \text{normal}(d, t_2) \land \\
\forall t_1, t_2 : t_2 \geq t_1 + DH \land (\neg \text{Danger}) \land \text{Danger during } [t_1, t_2] \rightarrow \\
\exists d, \Delta : d \leq t_1 + DH \land \Delta \leq t_1 + DH - d \land \text{normal}(d, \Delta) \land \neg \text{danger}(d, t_2).
\]

The pump activation unit switches the pump on or off before the specified deadline:

\[
\text{Pump} \equiv \forall t_1, t_2, \Delta : \\
on(t_1, \Delta) \land \neg \text{off}[t_1, t_2] \rightarrow (\text{pump} = \text{on}) \land \text{on}[t_1 + \Delta, t_2] \land \\
off(t_1, \Delta) \land \neg \text{on}[t_1, t_2] \rightarrow (\text{pump} = \text{off}) \land \text{off}[t_1 + \Delta, t_2].
\]

The control object decides whether it should send the message “on” or “off”, based on the information from the sensors:

\[
\text{Control} \equiv \forall d_1, d_2, t, \Delta_1, \Delta_2 : \\
(\text{danger}(d_1, \Delta_1) \land \neg \text{normal}(d_1, d_2) \rightarrow \text{off}(d_1, \Delta_1) \land \neg \text{on}(d_1, d_2)) \land \\
(\text{low}(d_1, \Delta_1) \land \neg \text{high}(d_1, d_2) \rightarrow \text{off}(d_1, \Delta_1) \land \neg \text{on}(d_1, d_2)) \land \\
(\text{high}(d_1, \Delta_1) \land \neg \text{low}(d_1, t) \land \text{normal}(d_2, \Delta_2) \land \neg \text{danger}(d_2, t) \rightarrow \\
(d_1 < d_2 \rightarrow \text{on}(d_2, \Delta_2) \land \neg \text{off}(d_2, t)) \land \\
(d_2 \leq d_1 \rightarrow \text{on}(d_1, \Delta_1) \land \neg \text{off}(d_1, t))).
\]

The detailed description of constituent objects of the control system can be found in [vdLaan00].

The presented decomposition is very convenient in a sense that checking its correctness is straightforward: conjunction the specifications of all parts of the system, immediately implies the top-level specification.

Although the verification of this decomposition is easy, the specifications of the constituent components are hard to implement. For example, the program implementing the control object has to generate the message \text{off} if the message \text{danger} is delivered and no \text{normal} messages are delivered within the specified interval. Because the specification of the control object does not contain ordering information of messages, the program cannot decide whether it is possible to receive \text{normal} message after it received a \text{danger} message.
Because of this problem, another specification style for the constituent objects was chosen. This specification style leads to the decomposition presented in section 4 of this report. It focuses not on timestamped intervals but on the time order of the messages between the objects.

### 5.1.5. The generic object specification framework

By means of the mine pump case study, we came to the very simple yet useful style for the specification of objects.

Suppose, the following simple control system (Figure 10) has to be designed:

![Figure 10. The simple mine pump control system.](image)

The input port “water” delivers messages from the water sensor. Messages from the water sensor may contain the data values “low” and “high”, representing the low and high water level measurements. The component must switch the mine pump off by sending the message containing “off” from the output port “command” in case of the low water level. If the water level is high, the mine pump has to be switched on by sending the “on” message to the output port “command”.

This system can be specified by the following assertions \((t, t_0 \in \text{TIME})\):

\[
\begin{align*}
\text{LowReaction} &\equiv \forall t_0 : \text{low}(t_0) \rightarrow \exists t : t_0 < t \land \text{off}(t); \\
\text{LowReason} &\equiv \forall t : \text{off}(t) \rightarrow \exists t_0 : t_0 < t \land \text{low}(t_0); \\
\text{HighReaction} &\equiv \forall t_0 : \text{high}(t_0) \rightarrow \exists t : t_0 < t \land \text{on}(t); \\
\text{HighReason} &\equiv \forall t : \text{on}(t) \rightarrow \exists t_0 : t_0 < t \land \text{high}(t_0); \\
\text{Specification} &\equiv \text{LowReaction} \land \text{LowReason} \land \text{HighReaction} \land \text{HighReason}.
\end{align*}
\]

The first assertion, \(\text{LowReaction}\), specifies that if the \textit{low} message is delivered to the port \textit{water}, the message \textit{off} has to be sent to the port \textit{command}. The second assertion, \(\text{LowReason}\), ensures that no “junk” \textit{off} messages are generated. It states that every outgoing \textit{off} message has a good reason, namely, the delivered \textit{low} message. The assertions \(\text{HighReaction}\) and \(\text{HighReason}\) describe a similar relationship between incoming \textit{high} messages and outgoing \textit{on} messages. The complete specification of the control system is a conjunction of these four assertions.

The case study showed that the order of events plays a great role in an asynchronous (distributed) system. The reason is that most events are not directly bound by a deadline and arrive thus in arbitrary order. Since the philosophy of this research project is “postponing the design decisions as much as possible”, the time order of the events in the system is specified as relaxed as possible.

The relaxed order means that we only define a time order for events that are also logically ordered. In this way, maximal parallelism can be achieved. For example, the events related to the same input port are usually meant to be ordered.
The “reaction/reason” specification style and the relaxed time order concept, described in this section, are used in the generic object specification framework, which is described in section 4.4.5.2.

5.1.6. Future extensions of the object specification framework

5.1.6.1. Stateless specification

In this section, we consider the design of the simple control system described in section 5.1.5.

The specification of the simple control system can be presented as the following state machine (Figure 11):

```
Figure 11. The state machine specifying the simple mine pump control system.
```

As we can see, this system can generate several subsequent `off` messages if subsequent `low` messages are delivered. Since the original description of the control system does not contain any restrictions for subsequent `off` messages, this specification already shows a design decision: several subsequent messages of the same type are allowed.

5.1.6.2. Specification using states

We can propose another specification for the control system from section 5.1.5, which does not allow subsequent messages of the same type. In order to do that, we introduce a notion of state into the assertions \((t, t_0 \in \text{TIME})\):

\[
\begin{align*}
\text{AltLowReaction} & \equiv \forall t_0 : \text{low}(t_0) \land (@ \text{On})(t_0) \rightarrow \exists t : t_0 < t \land (@ \text{Off})(t) \land \text{off}(t) ; \\
\text{AltLowReason} & \equiv \forall t : \text{off}(t) \lor (@ \text{Off})(t) \rightarrow \exists t_0 : t_0 < t \land (@ \text{On})(t_0) \land \text{low}(t_0) ; \\
\text{AltHighReaction} & \equiv \forall t_0 : \text{high}(t_0) \land (@ \text{Off})(t_0) \rightarrow \exists t : t_0 < t \land (@ \text{On})(t_0) \land \text{on}(t) ; \\
\text{AltHighReason} & \equiv \forall t : \text{on}(t) \lor (@ \text{On})(t) \rightarrow \exists t_0 : t_0 < t \land (@ \text{Off})(t_0) \land \text{high}(t_0) ; \\
\text{AltSpecification} & \equiv \\
\text{AltLowReaction} \land \text{AltLowReason} \land \text{AltHighReaction} \land \text{AltHighReason} .
\end{align*}
\]

The notation \((@ S)(t)\) used here means being in state \(S\) at moment \(t\) under assumption that the system cannot be in both states \(\text{On}\) and \(\text{Off}\) at the same time.

The first assertion of the alternative specification, \(\text{AltLowReaction}\), states that the system generates the message `off` and passes to the state `Off` if the message `low` has been accepted and the system stayed in the state `On`. The second assertion, `AltLowReason`, states that sending of the message `off` and changing the state to `Off` may happen only if the system, being in the state `On`, accepts the message `low`. The third assertion, `AltHighReaction`, is similar to `AltLowReaction`, and `AltHighReason` is similar to `AltLowReason`. The complete specification again is a conjunction of the four assertions.
The state machine given in Figure 12 visualizes this specification.

![State machine diagram](image)

Figure 12. The alternative specification of the simple mine pump control system.

5.1.6.3. Generalization

It is easy to see that the both specifications, presented in sections 5.1.6.1 and 5.1.6.2, are not equal. For example, the trace \(< \text{high, on, high, on}>\) is allowed by the state machine from Figure 11 but is not allowed by the state machine from Figure 12. And vice versa, the trace \(< \text{high, on, high, high}>\) is allowed by the state machine from Figure 12 but is not allowed by the state machine from Figure 11. This clearly shows that the presented state-machine specifications do not imply each other and, therefore, no one of them can be considered as more general than the other one. In other words, both specifications already contain design choices. This does not fit into our development methodology, where the design choices are postponed as long as possible.

Clearly, the “more general” top-level specification must imply all the presented specifications. One way to achieve this is to combine more general specification from two parts: what the system must do under certain conditions and what the system is allowed to do.

In our case, the “must” part of the specification is hidden in the alternative specification of the system because according to this specification, the system generates as few output messages as possible. On the other side, the first specification describes a behavior that allows the generation of many subsequent messages of the same type. It seems as a good choice for the “allowed behavior”. So, a mix of the both specifications results in a third one \((t, t_0 \in \text{TIME})\):

\[
\begin{align*}
\text{GenOffReason} & \equiv \forall t : (@ \text{ Off})(t) \rightarrow \exists t_0 : t_0 < t \wedge (@ \text{ On})(t_0) \wedge \text{low}(t_0); \\
\text{GenOnReason} & \equiv \forall t : (@ \text{ On})(t) \rightarrow \exists t_0 : t_0 < t \wedge (@ \text{ Off})(t_0) \wedge \text{high}(t_0); \\
\text{GenSpecification} & \equiv \\
& \quad \text{AltLowReaction} \wedge \text{LowReason} \wedge \text{GenOffReason} \wedge \\
& \quad \text{AltHighReaction} \wedge \text{HighReason} \wedge \text{GenOnReason}.
\end{align*}
\]

The two new assertions, \(\text{GenOffReason}\) and \(\text{GenOnReason}\), are needed here to prevent unprovoked changes of the system state.
The corresponding automaton is presented in Figure 13.

![Automaton Diagram](image)

Figure 13. The more general specification of the simple mine pump control system.

The automaton showed in Figure 14 represents a further generalization of the control object. The only difference between the automata showed in figures 13 and 14 is that the latter one can sporadically generate \textit{on} events if the active state is \textit{On} and the \textit{off} events if the active state is \textit{Off}. The decision, which design is preferable, may depend on other criteria than “generality”. For example, if load of communication channels used in the system has to be minimized, the design presented in Figure 13 may be better than the other.

![Automaton Diagram](image)

Figure 14. “Sporadic” control object.

The object specification framework described in section 4.4.5.2 implements a number of ideas on general specification patterns and ordering of actions, which are weaker than the sequential action order in state machines. In order to implement the ideas presented in this section, the formalism has to be extended by the state concept.

5.2. Conclusions

We have presented a formal compositional framework for the specification and verification of distributed real-time systems. Since the graphical notation used for specification of objects is similar to UML-RT, there is a strong connection between the presented framework and UML. But contrary to existing development methods, our approach supports the use of end-to-end timing constraints throughout the development process. No artificial timing constraints, that might affect the feasibility of the system, must be introduced during the decomposition of the system. In addition, these end-to-end timing constraints can be specified and logically verified independently from the execution platform. Only in a second phase of the development process, the objects are deployed on a
particular hardware configuration and the feasibility of the timing constraints is checked by a schedulability analysis (e.g., a rate-monotonic analysis for periodic systems) or by constructing a schedule. For this purpose, of course, the execution times of the various services the terminal objects provide at their interfaces must be estimated (e.g., during the architecting phase) or measured if the implementation is available.

Since the language supports timeliness and early verification, it can also be used as an architectural description language for real-time systems. Since our objects can also be considered as components, our object model also supports component-based architecting. The big advantage of this approach is that the components are fully specified and formal verification can already take place during the architecting phase.

The presented framework was used to verify a simple real-time control system with end-to-end timing constraints. This case study demonstrated the problem of temporal inaccuracy in real-time systems. A solution to this problem was suggested and its correctness was formally verified.
6. References


http://www.rational.com/products/robert


http://www.rational.com/uml


## Appendix A. The list of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle (A) \rangle { (C) }</td>
<td>The interface specification, where ( I ) is an interface, ( A ) is an assumption, ( C ) is a commitment</td>
<td>( \langle (A) \rangle { (C) } )</td>
</tr>
<tr>
<td>accept</td>
<td>((\text{PORTS} \times \text{D} \times \text{TIME})^p)</td>
<td>The message acceptance event</td>
</tr>
<tr>
<td>accept</td>
<td>((\text{PORTS} \times \text{D})^p)</td>
<td>The message acceptance event (abbreviation)</td>
</tr>
<tr>
<td>accept</td>
<td>(\text{PORTS}^p)</td>
<td>The message acceptance event (abbreviation)</td>
</tr>
<tr>
<td>B</td>
<td>{true, false}</td>
<td>The set of boolean values</td>
</tr>
<tr>
<td>D</td>
<td>\text{The set of data values}</td>
<td></td>
</tr>
<tr>
<td>DR</td>
<td>{\text{TIME}^p}</td>
<td>The set of device registers</td>
</tr>
<tr>
<td>during</td>
<td>(P \text{ during } I \equiv \forall t \in I : P(t))</td>
<td>The abbreviation</td>
</tr>
<tr>
<td>I</td>
<td>(\subseteq \text{TIME})</td>
<td>Time interval</td>
</tr>
<tr>
<td>inside</td>
<td>(P \text{ inside } I \equiv \exists t \in I : P(t))</td>
<td>The abbreviation</td>
</tr>
<tr>
<td>p</td>
<td>(p \in \text{PORTS})</td>
<td>Port</td>
</tr>
<tr>
<td>P</td>
<td>(B^{\text{TIME}})</td>
<td>The set of timed predicates</td>
</tr>
<tr>
<td>P</td>
<td>(P \in P)</td>
<td>Timed predicate</td>
</tr>
<tr>
<td>PORTS</td>
<td>{D^p}</td>
<td>The set of ports</td>
</tr>
<tr>
<td>read</td>
<td>((\text{DR} \times \text{D})^p)</td>
<td>The read device register event</td>
</tr>
<tr>
<td>read</td>
<td>(\text{DR}^p)</td>
<td>The read device register event (abbreviation)</td>
</tr>
<tr>
<td>t</td>
<td>(t \in \text{TIME})</td>
<td>Time moment</td>
</tr>
<tr>
<td>TIME</td>
<td>({t \in R \mid t \geq 0})</td>
<td>The time domain.</td>
</tr>
<tr>
<td>tmsg</td>
<td>((\text{D} \times \text{TIME})^{\text{TTimedMessage}})</td>
<td>Constructor of a timestamped message</td>
</tr>
<tr>
<td>v</td>
<td>(v \in \text{D})</td>
<td>Data value</td>
</tr>
<tr>
<td>write</td>
<td>((\text{DR} \times \text{D})^p)</td>
<td>The write device register event</td>
</tr>
<tr>
<td>write</td>
<td>(\text{DR}^p)</td>
<td>The write device register event (abbreviation)</td>
</tr>
</tbody>
</table>
Appendix B. PVS overview (based on [PVS])

PVS [OSR92, PVS] is a Prototype Verification System for specifying and verifying digital systems.

PVS consists of a specification language, a number of predefined theories, a theorem prover, and various utilities. PVS exploits the synergy between a highly expressive specification language and powerful automated deduction; for example, some elements of the specification language are made possible because the typechecker can use theorem proving. This distinguishing feature of PVS has allowed perspicuous and efficient treatment of many examples that are considered difficult for other verification systems.

The specification language of PVS is based on classical, typed higher-order logic. The base types include uninterpreted types that may be introduced by the user, and built-in types such as the booleans, integers, reals, and the ordinals up to epsilon_0; the type-constructors include functions, sets, tuples, records, enumerations, and recursively-defined abstract data types, such as lists and binary trees. Predicate subtypes and dependent types can be used to introduce constraints, such as the type of prime numbers. These constrained types may incur proof obligations during typechecking, but greatly increase the expressiveness and naturalness of specifications. In practice, most of the obligations are discharged automatically by the theorem prover. PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. Definitions are guaranteed to provide conservative extension; to ensure this, recursive function definitions generate proof obligations. Inductively-defined relations are also supported. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quantifiers, within a natural syntax. Names may be freely overloaded, including those of the built-in operators such as AND and -. Tabular specifications of the kind advocated by Parnas are supported, with automated checks for disjointness and coverage of conditions. An extensive prelude of built-in theories provides hundreds of useful definitions and lemmas; user-contributed libraries provide many more.

The PVS theorem prover provides a collection of powerful primitive inference procedures that are applied interactively under user guidance within a sequent calculus framework. The primitive inferences include propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. The implementations of these primitive inferences are optimised for large proofs: for example, propositional simplification uses BDDs, and auto-rewrites are cached for efficiency. User-defined procedures can combine these primitive inferences to yield higher-level proof strategies. Proofs yield scripts that can be edited, attached to additional formulas, and rerun. This allows many similar theorems to be proved efficiently, permits proofs to be adjusted economically to follow changes in requirements or design, and encourages the development of readable proofs.

A PVS specification consists of a collection of theories. Each theory consists of a header, which defines a theory name, theory parameters between square brackets, and a body, which is bracketed by keywords “BEGIN” and “END”. A body may contain declarations of types, variables, and formulas.

Type declarations are used to introduce new type names and to provide abbreviations for type expressions. A type declaration can be identified by the keyword “TYPE” following the new type identifier.

Variable declarations introduce new variables and associate a type with them. These are logical variables, not program variables; they have nothing to do with states. A variable declaration can be recognized using the keyword “VAR” followed by the type name.

Formula declarations may introduce axioms and theorems. The identifier associated with the declaration may be referenced during proofs. The expression that forms the body of the formula is a boolean expression. Axioms are introduced with the keywords “AXIOM”; theorems may be introduced with any of the words “LEMMA” or “THEOREM”. Axioms are treated specially when a
proof is analyzed, in that they are not expected to have an associated proof. Otherwise they are treated exactly like theorems. A formula declaration may contain free variables, which are treated as universally quantified in this case.