# Explanation of a Problem Specification according to the FOREST–Approach*

Martin Kronenburg
Research Group Formal Methods and Deduction
University of Kaiserslautern
kronburg@informatik.uni-kl.de

June 4, 1999

## 1 Introduction

A problem specification according to the FOREST approach describes a specific system that should be developed. The intention of a problem specification is to bridge the gap between the customer on the one hand and the developer on the other hand. Generally, several discussions between customer and developer are necessary to yield a common perception of the desired system. This common perception is documented in a problem specification.

Finally, a problem specification contains a description of the system as it *is*, e.g. the already existing installation or physical laws, as well as a description of the system as it *should be*, e.g. that the temperature in a room has to be always greater than 5 °C.

A problem specification according to the FOREST approach is not written in a way that it can be read as a whole from the beginning to the end; rather it is written in the style of a reference book. To be able to find specific parts more easily and more quickly a problem specification is structured and accompanied with some navigation support. The structure of a problem specification is illustrated in Section 2. The navigation support is explained in Section 3.

## 2 Structuring Concepts

A problem specification consists of a *Global Signature* standing at the beginning of the document and a set of *Description Classes*. The description class at the end of the document is called *Main Description Class* (cf. Figure 1).

In the following we introduce in detail the structure of a description class. Global signature (cf. Section 2.2) and main description class (cf. Section 2.3) are explained subsequently as special cases of a description class.

*This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Sonderforschungsbereich (SFB) 501, "Development of Large Systems with Generic Methods".

**Problem Specification**

**Global Signature**

**Description Class** $<\mathbf{name}_1>$

⋮

**Description Class** $<\mathbf{name}_n>$

**Main Description Class** $<\mathbf{name}>$

Figure 1: Structure of a problem specification

## 2.1 Description Class

In a description class a specific part of a system is described, e.g. a sensor or the light control of a room. A description class has an unambiguous name and is divided into six sections, cf. Figure 2.

**Description Class** $<\mathbf{name}>$
INTENTION
FORMAL PARAMETERS
BASE CLASSES
SIGNATURE
DOMAIN KNOWLEDGE
REQUIREMENT SPECIFICATIONS
MACHINE SPECIFICATIONS

Figure 2: Structure of a Description Class

Besides the INTENTION section every section can be omitted. But at least one of the other six sections has to exist in each description class.

### 2.1.1 Section INTENTION

This section has to exist in each description class. Here it is described briefly and informally, which part of the desired system is considered in this description class.

### 2.1.2 Section SIGNATURE

In this section the designations are introduced that are used to talk about the part of the system considered in this description class. Consider for example a description class dealing with the temperature control of a room. In such a description class one has to talk about the current room temperature, the settings of the radiator valves or whether a person in the room or not. In order to be able to talk about these phenomena of the real world corresponding designations have to be introduced.

Depending on the way one wants to talk about a phenomenon, a designation of the following seven kinds is used.

- Domain
- Function

- Timed Function
- Predicate
- Timed Predicate
- Time Constant
- Object

In the following we explain which kind of phenomenon can be represented by which kind of designation and we illustrate each kind by an example. Each designation is introduced in a SIGNATURE section using a tabular–like structure.

#### 2.1.2.1 Domain

A *domain* comprises a set of values. An example of a domain is the set of values a temperature sensor can deliver. In order to be able to talk unambiguously about this phenomenon, i.e. about this set of values, the designation *TEMP* is introduced as follows:

| | |
|---|---|
| **Domain** $TEMP = \{-40 \ldots 70\} < \text{INTEGER}$ | |
| Intention | : This domain contains the possible temperature values in degrees Celsius that can be delivered by a temperature sensor. |

**Example: Domain**

The entry in the field *Intention* explains the phenomenon represented by this designation. The formal part of the definition specifies the set of possible values, in the example -40 to 70. Using the symbol $<$ it can be specified whether the new domain is a subset of an already introduced domain. In the example the new domain *TEMP* is specified as a subset of *INTEGER*. This excludes that a temperature sensor delivers a real value, e.g. 4.11 °C. It can deliver only 4 or 5°C.

The following designations of domains do not have to be introduced explicitly, but can be used without being previously specified.

**NAT** This designation represents the set of natural numbers.

**INTEGER** This designation represents the set of integers.

**REAL** This designation represents the set of real numbers.

**TIME** This designation represents the set of all possible time points. This set is the set of the non–negative real numbers.

**STRING** This designation represents the set of all finite words over the alphabet $\{a, \ldots, z, A, \ldots, Z, \_, 0, \ldots, 9\}$.

The designations of domains are always written in capitals.

#### 2.1.2.2 (Timed) Function

A function assigns values of one domain to values of another domain. In the following example the function *tempVal* represents the value delivered by a temperature sensor at a specific time point. This means that the function *tempVal* assigns to each time point a value of the domain *TEMP*.

Most of the functions used in a problem specification concerning reactive systems represent phenomena that evolve over time, as for example the delivered value of a temperature sensor. In order not to have to mention always explicitly the domain *TIME*, such functions are called *timed functions*. Thus, in the following example **Timed Function** *tempVal* $() \to TEMP$ is an abbreviation for **Function** *tempVal* $(TIME) \to TEMP$.

| | |
|---|---|
| **Timed Function** $tempVal() \rightarrow TEMP$ | |
| Intention | : This function represents the value returned by a temperature sensor. |
| Scope | : environment controlled, visible for the machine |

**Example: Timed Function**

The entry in the field *Intention* explains the phenomenon represented by this designation.

To understand the entry in the field *Scope* some remarks are necessary. We suppose that a system can be split into the *environment* and the *machine*. The environment is the part of a system that already exists. Installed sensors or physical phenomena such a the temperature in a room belong to the environment. The machine is the part of a system that still has to be developed. An algorithm computing the settings of radiator valves for example belongs to the machine.

We suppose that each phenomenon is *controlled* by exactly one of these two parts. Control means that this part of the system controls the (not–) changing of a phenomenon. A phenomenon that is controlled by one of the two parts is also visible for this part. For the not controlling part a phenomenon can but do not have to be visible.

For example, the temperature in a room is controlled by the environment and is not visible for the machine. If a temperature sensor is already installed in the room, the value delivered by this sensor is also controlled by the environment but it is also visible for the machine. A radiator valve that is already installed in the room is normally controlled by the machine and visible for the machine. Values that the machine computes are phenomena controlled by the machine and which are not visible for the environment. Such phenomena are only allowed in a problem specification if they are explicitly introduced by the customer.

Due to this classification (control and visibility) one of the following four *scopes* is assigned to every phenomenon and by this to each function (and also to each predicate, cf. Section 2.1.2.4) representing a phenomenon:

- **e**nvironment controlled, not visible (**h**idden) for the machine (eh)
- **e**nvironment controlled, **v**isible for the machine (ev)
- **m**achine controlled, not visible (**h**idden) for the environment (mh)
- **m**achine controlled, **v**isible for the environment (mv)

#### 2.1.2.3 Time Constant

Besides the timed functions there is further special kind of functions occurring so often in a problem specification that it gets its own name: *time constants*. A time constant is a function that represents exactly one element of the domain *TIME*. Thus, in the following example **Time Constant** *durWinOpen* is an abbreviation for **Function** *durWinOpen* $() \rightarrow TIME$.

| | |
|---|---|
| **Time Constant** *durWinOpen* | |
| Intention | : This time constant represents the duration a window has to be open during one hour. |
| Scope | : environment controlled, not visible for the machine |

**Example: Time Constant**

#### 2.1.2.4 (Timed) Predicate

Often it is sufficient to know about a phenomenon whether it is valid or not. Considering for example a room one is only interested whether the room is empty or not. It is not important how many persons are currently in the room. For representing such 0-1-phenomena, we use *predicates*. A predicate is true (valid) or false (not valid).

Analogously to functions most of the predicates depend on time. Such predicates are called *timed predicates*. Thus, in the following example **Timed Predicate** *roomEmpty* is an abbreviation for **Predicate** *roomEmpty* (*TIME*).

| **Timed Predicate** *roomEmpty* | |
|---|---|
| Intention | : This predicate is true if and only if no person is the room. |
| Scope | : environment controlled, not visible for the machine |

**Example: Timed Predicate**

The meaning of the entries in the fields *Intention* and *Scope* is the same as for functions.

#### 2.1.2.5 Object

Due to the structuring into description classes it is possible to aggregate an instance of a description class in a different description class. Is there for example a description class *MotionDetector* describing a motion detector, it is possible to aggregate one or more instances of this description class in a description class *RoomTemp* specifying the temperature control of a room.

| **Object** *md* : *MotionDetector* | |
|---|---|
| Intention | : This object represents the room motion detector. |

**Example: Object**

The aggregation of an object means that all designations occurring in the aggregated class can — slightly modified — also be used in the aggregating class. The modification consists of adding the designation of the object and a "." before each designation of the aggregated class. Is there for example a designation *measuredEntity* in the description class *MotionDetector* then in the description class *RoomTemp* there is a designation *md.measuredEntity*.

Moreover, all statements of the aggregated class occurring in the sections DomainKnowledge (see Section 2.1.3) and Requirements/Specifications (see Section 2.1.4) are also valid in the aggregating class. The designations used in these statements are extended as described above.

### 2.1.3 Section Domain Knowledge

In this section of a description class the considered part of a system is described as it *is*. This means that the behavior of the system is described as it is without or despite of the machine. For example, the relationship between the measured temperature in a room and the temperature actually existing in the room belongs to the domain knowledge if the sensor is already installed in the room. Usually there are delays and deviations. The following example represents such a relationship.

| **Property** $D\_S_1$ | |
|---|---|
| Formal | : $\Box(envEntity \rhd^{modifyDuringMeas}_{\leq reactionTime} measEntity)$ |
| NL | : The measured value of a sensor (*measEntity*) is always derived from the value in the real world (*envEntity*) using the function *modifyDuringMeas*. A change of the value in the real world is always propagated within the time *reactionTime*. |

**Example: Domain Knowledge**

Every statement that belongs to the domain knowledge has a unique name standing after the keyword **Property**. In the example above this is $D\_S_1$. Every statement is given formally, i.e. as a temporal logic formula (after the keyword **Formal**) and in natural language (after the keyword **NL**).

### 2.1.4 Sections REQUIREMENT SPECIFICATIONS, MACHINE SPECIFICATIONS

In these sections of a description class the considered part of a system is described as it *should be*. This means that the behavior of the system is described as it should be with the machine. The structure of such a statement is the same as for a statement in the domain knowledge part.

The difference between the two sections REQUIREMENT and MACHINE SPECIFICATIONS is that a property in the REQUIREMENT SPECIFICATIONS section can not be implemented directly. More formally a requirement specification property has to contain at least one function or predicate symbol having the scope *eh* (see Section 2.1.2.2).

In contrast to this properties in MACHINE SPECIFICATIONS section have to be implementable. This means a machine specification property has to contain only function and predicate symbols that are visible for the machine, i.e. having the scopes *eh*, *mv*, or *mh*.

The following example illustrates the difference. Property *R_AL3* is a requirement specification since *safeLight* has scope *eh*. It is not possible for the machine to establish safe light directly. Property *M_AL7* is the corresponding machine specification. Here the previous predicate symbol *safeLight* is replaced by the actuators that are controlled directly by the machine.

REQUIREMENT SPECIFICATIONS

**Property** *R_AL3*

Formal : $\Box(\neg noSaveLight \wedge occ.occUsed \Rightarrow_{\leq T\_AL1} safeLight)$

NL : Whenever safe light has to be established and the area is occupied for at least $T\_AL_1$ milliseconds, the light is sufficient to move safely within this time span and remains sufficient to move safely at least as long as the precondition is true.

MACHINE SPECIFICATIONS

**Property** *M_AL7*

Formal : $\Box(\neg noSaveLight \wedge occ.occUsed \rightarrow \forall n \in NAT(numLights) : (evalSaveLight(n) = 1 \wedge light[n].ls.usedValue = 0 \rightarrow \Diamond_{\leq T\_AL2} light[n].pulse.noMalAct.adjustedEntity = 1))$

NL : Whenever safe light has to be established and the area is occupied the necessary actuators, i.e. pulse lines, are set within $T\_AL_2$ milliseconds to establish safe light.

**Example: Requirement and Machine Specifications**

There are also some requirements not dealing with the behavior of a system and which can not be formalized using the introduced temporal logic. Such requirements are also stated in the REQUIREMENTS/SPECIFICATIONS section using the keyword *NonFormal*. In these cases only a natural language version of the requirement is given.

**Property** $P\_LCP_1$

NonFormal : The control panel should be easy and intuitive to use.

**Example: Nonformalizable Requirement**

### 2.1.5 Section FORMAL PARAMETERS

In order to be able to define description classes more generally, it is possible to use parameters. In a problem specification there are two kinds of parameters: *domain parameters* and *number parameters*. Both kinds are illustrated using an example.

Consider different kinds of sensors, for example a temperature and a light sensor. They differ in the values they measure and deliver. Nevertheless, such sensors share common properties. To present such properties only once a general class *Sensor* is introduced. Special sensors like temperature or light sensor can then be *derived* from this general class, cf. Section 2.1.6. In the general description class *Sensor* the domains are only introduced but not specified. In the following example these are the domains *ENV_DOMAIN* and *MEASURED_DOMAIN*.

> **Description Class Sensor**
>
> INTENTION
> This class provides all definitions to describe the behavior of a *sensor*. ...
>
> FORMAL PARAMETERS
> **Domain** *ENV_DOMAIN*
> Intention : This domain contains the possible values of the measured phenomenon existing in the real world.
>
> **Domain** *MEASURED_DOMAIN*
> Intention : This domain contains the possible measured values that are returned by the sensor.
>
> ⋮

**Example: Domain Parameter**

Let us now consider an example for number parameters. Usually different rooms differ in the number of windows they have. In order to be able to describe common properties of rooms this number is expressed as a parameter. In the following example this is the parameter *numWin*.

> **Description Class Room**
>
> INTENTION
> This class provides all definitions to describe the behavior common to any kind of room. ...
>
> FORMAL PARAMETERS
> **Function** $numWin() \rightarrow NAT$
> Intention : This function represents the number of windows that belong to a room.
>
> ⋮

**Example: Number Parameter**

### 2.1.6 Section BASE CLASSES

As already mentioned a description class can be derived from another one. Consider for example a description class of a binary sensor. This is a sensor that can deliver the values 0 and 1. For instance a motion detector is a binary sensor. The following example illustrates the derivation of a binary sensor from a general sensor (see the example of a domain parameter). The general description classes from which a more concrete class inherits are mentioned in the BASE CLASSES section.

**Description Class BinarySensor**

INTENTION
This class provides all definitions to describe the behavior of a *binary sensor*. A binary sensor is a sensor that can return a 0 or a 1.

BASE CLASSES
**Class** *Sensor*(*ENV_DOMAIN* = *BINARY*,
*MEASURED_DOMAIN* = *BINARY*)
Intention : All definitions of Sensor are also provided here using the values 0 and 1 as the possible values of the phenomenon existing in the real world and using the values 0 and 1 as the possible measured values.

⋮

**Example: Inheritance**

If a description class is derived from another one, the concrete class *inherits* all designations and properties of the general class. In the example above this means that all designations introduced in the description class *Sensor* can also be used in the description class *BinarySensor*. Moreover, all statements of the DOMAIN KNOWLEDGE and REQUIREMENTS/SPECIFICATIONS sections of the *Sensor* class are also valid in the *BinarySensor* class.

### 2.2 Global Signature

The global signature at the beginning of a problem specification is a description class that contains only the SIGNATURE section. The difference to other description classes is that the designations introduced in the global signature are visible for all other description classes. Hence, in a global signature such designations are introduced that can not be assigned to only one specific part of a system.

### 2.3 Main Description Class

The final specification of the system that should be developed is expressed in the *main description class* at the end of a problem specification. All other description classes support a compact and concise representation of the main description class. Usually in the main description class only the relevant objects are aggregated. The behavior of these objects is specified in the corresponding classes.

## 3 Navigation Support

In order to find a specific information more easily in a problem specification it contains besides the description classes the following parts.

### 3.1 Table of Contents

The table of contents at the beginning of a problem specification contains all description classes. Moreover, the description classes are grouped. Each group comprises several description classes that are closely related concerning a specific criterion. These groups support only the structuring but do not have any influence on the meaning of a problem specification.

## 3.2 Index

At the end of a problem specification there is an alphabetically sorted index. It contains all designations used in the problem specification and the names of all description classes.

## 3.3 Aggregation– and Inheritance–Hierarchy

Before the description classes are described an overview about the relationships among the description classes is presented by displaying the aggregation and inheritance hierarchies. This graphical overview consists of several figures. Each figure contains the relationships in one of the groups, see Section 3.1. The root of the first figure is the main description class. Starting from this class the relations are depicted until a class is not refined any more or until a class belongs to a different group. If a class of a different group is further structured this is represented in another figure.

If a description class $A$ aggregates $n$ objects of the description class $B$, this is depicted as follows:

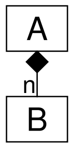


Figure 3: Graphical Representation of Aggregation

If a description class $D$ is derived from a description class $C$, i.e. $D$ inherits from $C$, this is depicted as follows:

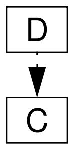


Figure 4: Graphical Representation of Inheritance

## 3.4 References

There are two kinds of references in a problem specification:

- pre–references
- post–references

A pre–reference point to the information in the original documents provided by the customer. A pre–reference can be assigned to a designation of the SIGNATURE section and to a property.

A post–reference can be assigned only to designations of a SIGNATURE section. It refers to the properties where the corresponding symbol is used.