Journal of Universal Computer Science, vol. 6, no. 7 (2000), 679-703
submitted: 30/12/99, accepted: 20/4/00, appeared: 28/7/00 © Springer Pub. Co.

Application of the FOREST Approach to the *Light Control Case Study*

Martin Kronenburg
(University of Kaiserslautern, Germany
kronburg@informatik.uni-kl.de)

Christian Peper
(University of Kaiserslautern, Germany
peper@informatik.uni-kl.de)

Abstract: FOREST is a requirements engineering approach designed to support the creation of precise and intelligible problem specifications of reactive systems. It integrates a product model, a process model, and an editing tool. In this paper, we present the results of applying the FOREST approach to the *Light Control Case Study*. This includes the presentation of excerpts of the resulting problem specification, as well as the discussion of the strengths and shortcomings of the FOREST approach.

Key Words: formal requirements specification, problem specification, real-time temporal logic, object-orientation

Category: C.3, D.2.1, D.2.2, F.3.1, F.4.3

1 Introduction

It is widely accepted that the first activities in the development of (software) systems are very critical because they have a great impact on the subsequent phases. Customer and system developer have to agree on a common perception of the problem that has to be solved by a system. This agreement should be documented in an intelligible and precise way.

FOREST[1] is an approach that has been designed to support the creation of precise and intelligible problem specifications, especially in the context of reactive systems. FOREST integrates a product model, a process model, and an editing tool. The *product model* provides a reference model for *problem specifications*, which comprise all information about *which* system to be built. The product model also provides an instantiation of this document type for reactive systems that is based on a real-time temporal logic and the structuring concepts modularization, aggregation, inheritance, and parameterization. The *process model* prescribes the steps to be followed by the system developer as well as the interaction between customer and developer when creating a problem specification.

[1] FOREST is an acronym for **Fo**rmal **Re**quirement **S**pecification **T**echnique.

The tool *xforest* supports the creation of syntactically correct specifications and the consistency of references.

The focus of this paper is to illustrate and discuss the results of applying the FOREST approach to the *Light Control Case Study (LCCS)*. We start with a brief survey of the FOREST approach in [Section 2]. [Section 3] presents an overview of the FOREST problem specification of the *Light Control Case Study*. In [Section 4], several excerpts of this specification are considered in more detail and used to explain the concepts of the FOREST approach. The complete problem specification is electronically available at [KP00]. In [Section 5], we analyze the problem specification, report on some statistical results, and elaborate on strengths and shortcomings of the FOREST approach. We end with some concluding remarks in [Section 6].

2 Survey of the FOREST Approach

In this section, we provide a brief survey of the FOREST approach. For more details concerning the various concepts realized in this approach, please refer to [PGK97], [GKP98], and [KP99]. The objectives and concepts outlined in the following are illustrated in [Section 3] and [Section 4], where we discuss the application of the FOREST approach to the LCCS.

2.1 Objectives

The FOREST approach has been designed with the following objectives:

Precision avoids ambiguities and thus potential conflicts between customer and system developer.

Intelligibility is important for the acceptance and maintainability of the document.

Expressiveness determines the degree to which a problem specification can be formalized.

Scalability opens the way for the treatment of large systems.

Reuse within and across projects can increase productivity and reduce costs.

Traceability supports the validation of completeness and correctness, and the effective propagation of changes into all depending specification parts.

Practicability is related to the effort to create a specification.

To point out which objectives are addressed in the following specification excerpts, and by which means they are supported in the FOREST approach, we have included references of the form *Objective by Means* in some of the headings in [Section 4] and [Section 5].

2.2 Reference Model

When creating a problem specification, the main goal is to obtain a perception of the desired system that is common to customer and system developer. A substantial precondition to achieve this goal is a precise characterization of what a problem specification should be and what it should contain. Therefore, the FOREST approach contains a general reference model for problem specifications [KP99]. This reference model prescribes the content of such a document independently of a particular description technique.

The FOREST reference model is based on the work of Jackson, Zave, et al. [ZJ97, GGJZ00]. A related reference model, usually called the *four variable model* was proposed by Parnas et al. [vSPM93, PM95]. Both reference models are compared in [GGJZ00]. The *CoRE* method [FBWK92] is closely related to the FOREST approach, and uses a similar reference model.

2.2.1 Basic Notions

A *system* is a part of the real world that is for some reason considered as a unit. In a system, several *phenomena* are collected and combined. A *phenomenon* is an aspect in the real world that is essential for a system. Phenomena are, for example, states of a system (e.g., in the LCCS the light intensity in a room), events occurring in the real world (e.g., the switching on of a light), objects (e.g., sensors and actuators), and individuals (e.g., a facility manager of a building). *Terms* (e.g., words in natural language) are used to designate specific phenomena of a system. They are a prerequisite for each conversation about real world phenomena. We demand that there is a one–to–one relation between phenomena and terms, i.e., for each phenomenon there is exactly one term designating it, and each term designates exactly one phenomenon. *Statements* express relationships between several phenomena. Each statement is constructed of the terms representing the considered phenomena. An example of a statement is: *If a room is occupied, then the light is on.* Here, a relation between the terms *room*, *occupied*, *light*, and *on* is expressed.

2.2.2 Classification of Phenomena

For the development of a system, it is very important to distinguish between the part of a system that already exists, the *environment*, and the part of a system that has still to be developed, the *machine*. Each phenomenon and also each term (recall the one-to-one relation) has to be uniquely assigned to either the environment or the machine. Although this is obviously a very central feature of a phenomenon/term, it is neglected in many approaches and not expressed explicitly. Let us consider the *LCCS*: while some sensors and actuators, such as motion detectors or dimmers, are pre-installed and thus parts of the environment, the control panels, for example, do not exist. They belong to the machine and have to be developed.

Based on this distinction between environment and machine, phenomena/terms are classified further w.r.t. the following two aspects:

control: It is demanded that each phenomenon/term is controlled either by the environment or by the machine.

visibility: It is demanded that each phenomenon/term controlled by one of these parts is also visible to this part.

We say that a phenomenon *belongs to* the part it is controlled by. Note that a phenomenon/term can be visible to both parts.

From these constraints concerning control and visibility, the following kinds of phenomena/terms can be derived:

eh: **e**nvironment controlled, **h**idden to the machine

ev: **e**nvironment controlled, **v**isible to the machine

mv: **m**achine controlled, **v**isible to the environment

mh: **m**achine controlled, **h**idden to the environment

eh, *ev*, *mh*, and *mv* denote the *scope* of a phenomenon/term. It is required that each phenomenon/term has exactly one of these scopes.

2.2.3 Classification of Statements

The distinction between what already exists and what still has to be developed has a great impact on the classification of the statements and descriptions. We demand that each statement has to be either in *indicative* mood or in *optative* mood. A statement in indicative mood describes something in the real world as it *is*. A statement in optative mood describes something in the real world as it *should be*.

When combining the mood of a statement with the scopes of the terms that occur in this statement, a large variety of different kinds of statements is possible, but only the following three disjoint combinations of mood and scope are reasonable:

domain statement: a statement in indicative mood where all terms are visible to the environment (scopes *eh*, *ev*, or *mv*).

requirement statement: a statement in optative mood where at least one term with scope *eh* is used.

machine statement: a statement in optative mood where all terms are visible to the machine (scopes *ev*, *mv*, or *mh*).

For more details concerning the reference model, please refer to [KP99].

2.3 Instantiation of the Product Model

Within the FOREST approach, the general reference model of problem specifications is instantiated using a real-time *temporal logic* (see [KPG96]) as basic formal description technique. This allows the creation of precise problem specifications with an unambiguous semantics. To enhance the intelligibility of a problem specification, we provide explanations in *natural language* for each elementary phenomenon, object, and formalized property. Moreover, to be able to handle problem specifications of large systems, we have combined *structuring* concepts such as *modularization*, *aggregation*, *inheritance*, and *parameterization* with the temporal logic. The resulting product model is explained and illustrated in [Section 3] and [Section 4].

2.4 Process Model

Besides this product model, the FOREST approach comprises a *process model* that prescribes an iterative interaction between customer and developer to create a problem specification. In this process model, also the application of *requirement patterns* is incorporated (see [GKP98]). For details, see [PGK97].

2.5 Tool Support

Finally, the editing tool *xforest* supports the creation of problem specifications according to the product model (details in [SS99]). It also maintains the consistency of *references* to achieve traceability.

3 Overview of the Final Problem Specification

The application of the Forest approach to the LCCS [PD00] resulted in a problem specification called LCCS-PS. By application of the tool *xforest*, representations in Postscript$^{\mathrm{TM}}$ and HTML format were generated. Based on a "vertical cut" through the LCCS-PS, we will elaborate on several objectives and concepts addressed by the Forest approach. In [Section 4], we provide an excerpt of the LCCS-PS (mainly of the Postscript$^{\mathrm{TM}}$ representation). We assume that the reader is familiar with the original problem description.

3.1 Document Structure

To manage problem specifications of *large* systems, we structure a problem specification into several so-called *description classes*. Relations among description classes are established by aggregation and inheritance. Description classes specifying common aspects are further organized in so-called *groups*. In the LCCS-PS we have the following eight groups and 44 description classes:

Datatypes This group contains the description class *LightScene*. In this description class, we formalize the information contained in the corresponding entry in the *Dictionary of Terms* of the original problem description.

BasicHumanMachineInterface This group only contains the description class *EnteredValue* where the basic interface between machine and persons is specified.

Sensors This group consists of all description classes that contain specifications related to sensors. It is discussed in [Section 4.1].

Actuators This group consists of all description classes that contain specifications related to actuators.

CombinedSensorsAndActuators This group contains all description classes in which several sensors and actuators are combined. It is discussed in [Section 4.2].

HumanMachineInterface This group contains the description classes where the interface between machine and persons is specified on a more abstract level. This includes the specifications of control panels.

Occupancy This group contains the description classes where occupancy of an area, i.e. a hallway section or a room, is defined.

BuildingUnits This group contains the description classes where the units of the floor, i.e. rooms, hallway sections, and staircases, as well as the so called *main description class Floor* are specified. These description classes are discussed in [Section 4.3] and [Section 4.4].

For each group containing more than one description class, we have created a diagram displaying the aggregation and inheritance relations among the description classes of this group (see [Fig. 1], [Fig. 4], and [Fig. 7]). In these figures, boxes with bold frames refer to description classes that belong to other groups. The shaded parts refer to the excerpts considered in more detail in [Section 4]. Not all groups, diagrams and other details could be addressed here. They can be found in [KP00].

3.2 Description Class Structure

We shortly explain how description classes are structured. In general, a description class consists of seven parts:

– Intention
– Formal Parameters
– Base Classes
– Signature
– Domain Knowledge
– Requirement Specification
– Machine Specification

The Intention part is mandatory. All other parts are optional. In the Intention part, a developer informally describes which part of a system is considered in a description class. For sake of brevity, the Intention part will be omitted in the following examples. Instead the intention of a description class will be explained in the text.

In the Formal Parameters part, two kinds of parameters of a description class can be specified: *sort parameters* and *number parameters*. A parameter of a parameterized description class always has to be instantiated if this class is aggregated or if another description class inherits from this parameterized description class. Both kinds of parameters are explained and illustrated below.

In the Base Classes part, all description classes from which this description class inherits are listed. The Base Classes parts of all description classes thus defines the *inheritance* relation among the description classes. Note that inheritance between two classes also implies a *specialization* between both classes, in the sense that the derived class is an extension of the base class.

The Signature part contains all entities that are necessary to talk about the part of the system considered in this description class. In the excerpts below, different kinds of entities appear. Following the guidelines of the reference model, there has to be an *intention* explaining which phenomenon is represented to support the one-to-one relationship between phenomena and terms. Moreover, the scope of each entity representing an "elementary phenomenon" has to be specified.

In the DOMAIN KNOWLEDGE, REQUIREMENT SPECIFICATION, and MACHINE SPECIFICATION parts, the properties of the objects belonging to a description class are specified. The properties are assigned to these three parts according to the classification of statements given by the reference model. The name of each property is chosen to reflect the description class and the part to which this property is assigned, e.g., *D_OLS2* denotes the second property in the DOMAIN KNOWLEDGE part of description class *OutdoorLightSensor* [Fig. 3]. Each property usually consists of a formula in real-time temporal logic and of a translation of this formula into natural language.

3.3 Development

We started with the specification of the description classes of the groups *Datatypes*, *BasicHumanMachineInterface*, *Sensors*, and *Actuators*. Next, we considered the description classes of the groups *CombinedSensorsAndActuators*, *HumanMachineInterface*, and *Occupancy*. Finally, the description classes of the group *BuildingUnits* were specified. Note that this bottom-up process could not be followed strictly, since the specification of a description class aggregating, for example, another led to modifications of the aggregated description class. Nevertheless, in principle the described bottom-up process was carried out.

4 Excerpts of the Final Problem Specification

We present the excerpts of the LCCS-PS in a bottom-up way that corresponds to the way the LCCS-PS is structured. This structure ensures that all entities are introduced before they are used. We will consider four different specification levels: in [Section 4.1], we start with the most elementary description classes and provide an excerpt of the group *Sensors* as an example. In [Section 4.2], we consider two examples of the group *CombinedSensorsAndActuators*, namely *CeilingLightGroup* and *DimmableCeilingLightGroup*. On the next description level, we consider several description classes of the group *BuildingUnits* in [Section 4.3]. Finally in [Section 4.4], we present an excerpt of the top description class *Floor*, the so-called *main description class*.

4.1 Group Sensors

[Fig. 1] shows a class diagram with aggregation and inheritance relations for the description classes of group *Sensors* in UML notation [BRJ99]. The two groups *Actuators* and *Occupancy* are structured in a similar way. The description classes *NoMalfunctionSensor*, *MalfunctionSensor*, *Sensor*, and *BinarySensor* are abstract description classes from which the four concrete description classes *OutdoorLightSensor*, *StatusLine*, *DoorClosedContact*, and *MotionDetector* are derived. In these description classes, we mainly formalize the information given in Section 2.8 of the original problem description [PD00].

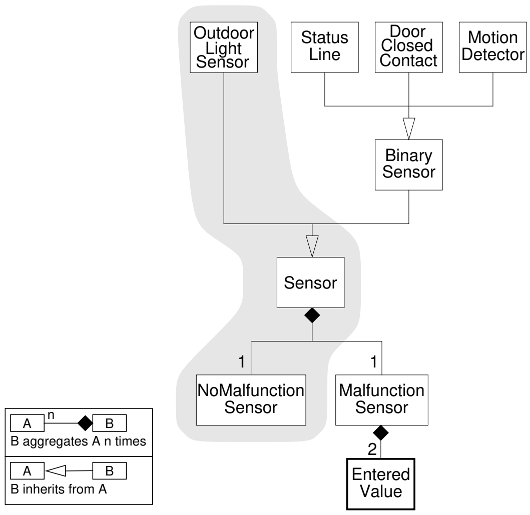

Figure 1: Class Diagram of Group *Sensors*

[Fig. 2] shows a leaf description class, namely the description class *NoMalfunctionSensor*. This description class generally specifies the behavior of a sensor in normal operation. This description class is aggregated by description class *Sensor* from which description class *OutdoorLightSensor* is derived. Hence, each outdoor light sensor *is a* sensor, and therefore it must contain an object that *is a NoMalfunctionSensor*.

4.1.1 NoMalfunctionSensor

In the description class *NoMalfunctionSensor*, three kinds of entities are introduced. *reactionTime* is a **Time Constant**. This means that it represents a real

Description Class NoMalfunctionSensor

FORMAL PARAMETERS

Sort *ENV_DOMAIN*

Intention : This domain contains the possible values of the measured phenomenon of the real world.

Sort *MEASURED_DOMAIN*

Intention : This domain contains the possible measured analog values.

Sort *CONVERTED_DOMAIN*

Intention : This domain contains the possible digital values, converted from an analog value.

SIGNATURE

Time Constant *reactionTime*

Intention : This time constant represents the *reaction time* of a sensor. ...

Scope : ev

⋮

Timed Function *envEntity* → *ENV_DOMAIN*

Intention : This function represents the value of the measured phenomenon in the real world.

Scope : eh

Timed Function *measuredEntity* → *MEASURED_DOMAIN*

Intention : This function represents the measured analog value of the phenomenon of the real world.

Scope : eh

Timed Function *convertedEntity* → *CONVERTED_DOMAIN*

Intention : This function represents the digital value derived from the measured analog value.

Scope : ev

Function *modifyReaction* : *ENV_DOMAIN* → *MEASURED_DOMAIN*

Intention: This function represents the way in which a value of the phenomenon of the real world (*envEntity*) is modified during the reaction of a sensor resulting in a measured analog value *measuredEntity*.

Scope : eh

⋮

DOMAIN KNOWLEDGE

Property *D_NMS1*

Formal : $\Box(\mathit{envEntity} \rhd^{\mathit{modifyReaction}}_{\mathit{reactionTime}} \mathit{measuredEntity})$

NL : The measured analog value is always derived from the value of the phenomenon of the real world using the function *modifyReaction*. A change of the value in the real world is always propagated within the time *reactionTime*.

⋮

Figure 2: Description Class *NoMalfunctionSensor*

number. The three **Timed Function**s *envEntity*, *measuredEntity*, and *convertedEntity* represent phenomena whose value can change over time. In contrast to this, the **Function** *modifyReaction* represents a phenomenon that is independent of time. SIGNATURE entities of these three kinds always represent “elementary phenomena” for which a scope has to be specified.

Precision by Scopes

Looking at the three timed functions *envEntity*, *measuredEntity*, and *convertedEntity*, we want to emphasize the significance of the *scope* assigned to a phenomenon. While the phenomena represented by *envEntity* and *measuredEntity* are not visible to the machine, the phenomenon represented by *convertedEntity* is visible to it. Note that the scopes would be different if the sensors do not yet exist. In this case, the scope of *envEntity* would be *ev*, and that of *measuredEntity* and *convertedEntity* *mh*. To distinguish between the different kinds of control and visibility is especially important in the description classes specifying actuators and the aspect *occupancy*.

Finally, we informally explain the meaning of the temporal operators $\Box$ and $\rhd$ we have applied in property *D_NMS1* to specify the delayed and modified dependence of an analog entity s_2 from another analog entity s_1. The meaning of a temporal formula $\Box\varphi$ is that φ holds always. $s_1 \rhd_\tau^f s_2$ means that s_2 has a value that is the result of function f applied to a value that s_1 has had sometime during the last τ time units.

4.1.2 OutdoorLightSensor ***Reuse and Scalability by Inheritance***

The description class *OutDoorLightSensor* is obtained by specializing *Sensor* and instantiating the sort parameters *ENV_DOMAIN*, *MEASURED_DOMAIN*, and *CONVERTED_DOMAIN* in the Base Classes part (see [Fig. 3]).

Description Class OutdoorLightSensor

Base Classes
Class *Sensor*(*ENV_DOMAIN* = *REAL*, *MEASURED_DOMAIN* = *REAL*,
CONVERTED_DOMAIN = *OUT_LIGHT_VALUES*)
Signature
Domain *OUT_LIGHT_VALUES* = $\{1 \ldots 10000\}$
Intention: This domain contains the possible values measured by an outdoor light sensor.

⋮

Domain Knowledge

⋮

Property *D_OLS2*
Formal : $\Box(noMalSens.reactionTime = 10)$
NL : The reaction time of an outdoor light sensor is 10 milliseconds.

⋮

Figure 3: Description Class *OutdoorLightSensor*

Furthermore, the specialization of the behavior can be expressed in the derived class by either making a certain property more restrictive or by adding properties. The reaction time of an outdoor light sensor, for instance, is restricted to a particular value in property *D_OLS2* in the Domain Knowledge part. This reflects the corresponding entry in Table 1 of the original problem description [PD00]. A dot notation is used to refer to attributes of aggregated classes.

4.2 Group CombinedSensorsAndActuators

The group *CombinedSensorsAndActuators* (see [Fig. 4]) comprises the description classes *CeilingLightGroup* and *DimmableCeilingLightGroup*. The description classes of this group formalize information contained in the paragraphs 7, 8.2, 13, 14.2, 19, and in Figure 3 of the original problem description. These description classes have in common that they aggregate several sensors and actuators and, thus, introduce a new level of abstraction that is also provided by the original problem description, cf. Paragraphs 7, 8, 13, and 14 as well as the *Dictionary of Terms* of the original problem description [PD00].

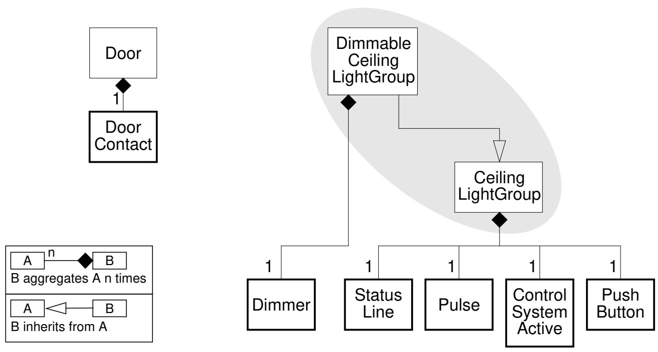

Figure 4: Class Diagram of Group *CombinedSensorsAndActuators*

4.2.1 CeilingLightGroup *Intelligibility and Reuse by Aggregation*

In the Forest approach, aggregation is expressed by an object declaration in the Signature part of the aggregating description class. Each object declaration starts with the keyword **Object**, followed by the name of the object and the name of an already specified description class to which this object shall belong. Since an object does not represent an elementary phenomenon, no scope has to be provided, but an intention in natural language must be given (see [Fig. 5] and [Fig. 6]).

Description Class CeilingLightGroup

FORMAL PARAMETERS

Function $numPB \rightarrow NAT$

Intention : This parameter represents the number of push buttons with direct control of the ceiling light group (bypassing the machine).

SIGNATURE

Object sl : *StatusLine*

Intention : This object represents the status line indicating whether a ceiling light group is on or off.

Object $pulse$: *Pulse*

Intention : This object represents the actuator of a ceiling light group the machine uses to switch the ceiling light group on or off.

Object csa : *ControlSystemActive*

Intention : This object represents the actuator of a ceiling light group the machine uses to signal that it is still active.

Object pb : $PushButton[numPB]$

Intention : These objects represent the push buttons of a ceiling light group a person can use to switch the light on or off. Note that a push of a button is not directly visible to the machine.

⋮

Figure 5: Description Class *CeilingLightGroup*

Description class *CeilingLightGroup* has the number parameter *numPB*. Such number parameters can be used on the one hand as functions representing a nonnegative integer. On the other hand they can be used to specify the size of an *object array*. In the description class *CeilingLightGroup*, *numPB* push buttons are aggregated. By this, we can use the description class *CeilingLightGroup* to model the ceiling light groups in a hallway section as well as in a room. Note that according to the original problem description [PD00], *each ceiling light group of a room is controlled by one push button* (Paragraph 8.2), and *each ceiling light group of a hallway section is controlled by several push buttons* (Paragraph 14.3).

4.2.2 DimmableCeilingLightGroup *Reuse by Inheritance*

The excerpt of the description class *DimmableCeilingLightGroup* illustrates a further possibility to specialize a description class – in addition to those mentioned in [Section 4.1]. Here, we introduce the new object *dimmer*, i.e. we extend the signature of class *CeilingLightGroup* (see [Fig. 5] and [Fig. 6]).

4.3 Group BuildingUnits

[Fig. 7] shows the aggregation and inheritance hierarchy among the description classes of the group *BuildingUnits*, which is the top level group.

Description Class DimmableCeilingLightGroup
BASE CLASSES
Class $CeilingLightGroup(numPB = 1)$
Intention : All definitions of description class *CeilingLightGroup* are inherited.
SIGNATURE
Object *dimmer* : *Dimmer*
Intention : This object represents the dimmer the machine uses to set the illuminance of a ceiling light group.

Figure 6: Description Class *DimmableCeilingLightGroup*

4.3.1 Area — ***Traceability by References***

In description class *Area* [Fig. 8], we specify the common properties of hallway sections and rooms. This includes, for example, property *R_A2* which is a formalization of the facility manager needs FM2 and FM3 of the original problem specification. The traceability relation between these needs and property *R_A2* is explicitly expressed by the additional entry:

Prereferences: pdref FM2
pdref FM3

In the complete LCCS-PS, such traceability information is added to each signature entry and to each specified property. For sake of brevity, we have omitted these references in the other excerpts.

Obviously, such traceability information has to be explicitly given by the developer. Nevertheless, our tool *xforest* supports this traceability relation in the following way: besides a problem specification, a developer also enters the original problem description in a general format [SS99]. Then it is possible to associate either with a property the needs that are formalized by this property or with a need the properties that formalize it. The corresponding inverse relation is computed by the tool. In the HTML representation of the LCCS-PS, this traceability relation is additionally supported by hyper-links (see [KP00]).

Besides the traceability relation between the original problem description and a problem specification the FOREST approach supports a further traceability relation within a problem specification. Let φ be a formula in a property and Ω_φ the set of symbols occurring in φ then for each symbol $\omega \in \Omega_\varphi$ there is a link to the signature entry where ω is introduced. Analogously, we associate with each signature entry a set of *postreferences* referring to the properties where this signature entry is used. This traceability relation between properties and signature entries is computed automatically. Note that due to the aggregation and inheritance relations, this computation is not trivial. In the case of the HTML representation this traceability relation is also supported by hyper-links.

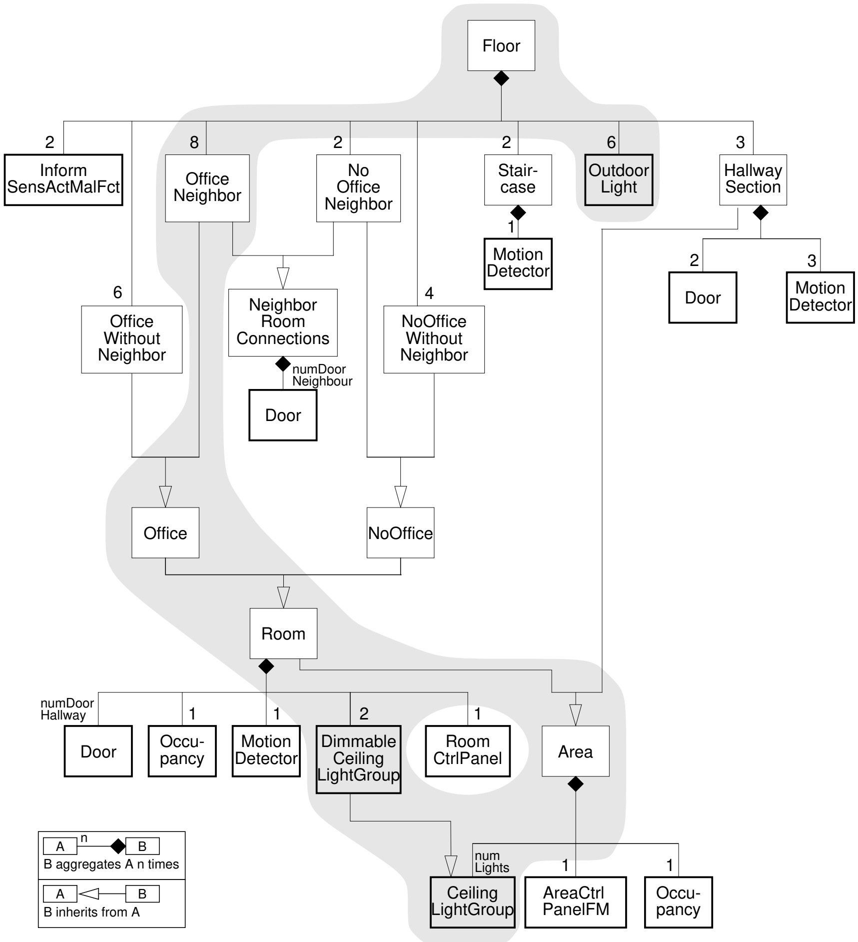

Figure 7: Class Diagram of Group *Building Units*

Precision and Reuse by Requirement Patterns

Considering property *R_A2* of description class *Area* [Fig. 8], we can illustrate two further aspects of the FOREST approach: the increase of *precision* and the *reusability* of *requirement patterns*. In the original problem description, the two needs FM2 and FM3 only state that the ceiling light groups in an area have to be off if an area is unoccupied for a certain time span. It is not stated that the ceiling light groups have to *remain* off. This situation is typical: on the one hand this additional aspect is often forgotten in a natural problem description and on the other hand a state has to remain valid as long as a certain precondition holds.

Description Class Area

FORMAL PARAMETERS

Function $numLights \rightarrow NAT$

Intention : This parameter represents the number of lights in an area that are controlled by the machine.

Function $numPB \rightarrow NAT$

Intention : This parameter represents the number of push buttons that can be used by a person to control a ceiling light group in an area by bypassing the machine.

SIGNATURE

⋮

Object $light : CeilingLightGroup[numLights]$

Intention : These objects represent the ceiling light groups in an area that are controlled by the machine.

⋮

REQUIREMENT SPECIFICATION

⋮

Property R_A2

Formal : $\Box(\neg occFM.occEnv \Rightarrow_{\leq T\_A1} \forall n \in \{1, \ldots, numLights\} : (light[n].sl.noMalSens.envEntity = 0))$

NL : Whenever an area is unoccupied for at least *T_A1* milliseconds, all ceiling light groups in this area are off within this time and remain off at least as long as the area is unoccupied.

Prereferences: pdref FM2
pdref FM3

⋮

Figure 8: Description Class *Area*

Since FM2 and FM3 are representative for a particular type of requirement that occurs quite frequently, it was possible to apply the *requirement pattern DelayedImplication*(φ, ψ) (see [Section 5.1]), yielding property *R_A2*. This pattern has been defined during a previous case study to normalize the specification of the corresponding properties.

In general, a requirement pattern consists of a formula, a translation of this formula into natural language, examples of instantiations, and theorems expressing valid properties of a requirement pattern. For more details concerning requirement patterns, please refer to [PGK97] and [GKP98]. In [Section 5.1], we report on some statistical results of applying requirement patterns in the LCCS-PS.

4.3.2 Room

Let us now discuss description class *Room* [Fig. 9] in more detail to illustrate, how product model and process model of the FOREST approach contribute to the precision and the completeness of specifications. We will also reconsider specialization.

Description Class Room

FORMAL PARAMETERS

Function *numDoorHallway* $\rightarrow$ *NAT*

Intention : This parameter represents the number of doors of a room to a hallway section.

BASE CLASSES

Class *Area*($numLights = 2, numPB = 1$)

Intention : All definitions of description class *Area* are inherited. There are two ceiling light groups in a room that are controlled by the machine:

- *light*[1] represents the ceiling light group near the window and
- *light*[2] represents the ceiling light group near the wall to a hallway section.

SIGNATURE

Object *Area* :: *light* : *DimmableCeilingLightGroup*

Intention : The ceiling light groups in a room are *dimmable* ceiling light groups.

Timed Function *curLightScene* $\rightarrow$ *LightScene*

Intention : This function represents the current light scene that is established by the machine if light scenes are enabled.

Scope : mh

Timed Function *curDefaultLightScene* $\rightarrow$ *LightScene*

Intention : This function represents the current default light scene that is established by the machine if light scenes are enabled.

Scope : mh

Timed Function *malDefaultLightScene* $\rightarrow$ *LightScene*

Intention : This function represents the light scene that is the default light scene in the case of a malfunction of the outdoor light sensors.

Scope : mh

Timed Function *noMalDefaultLightScene* $\rightarrow$ *LightScene*

Intention : This function represents the light scene that is the default light scene in the case that the outdoor light sensors have no malfunctions.

Scope : mh

MACHINE SPECIFICATION

Property *I_M_R1*

Formal : $curDefaultLightScene = noMalDefaultLightScene$

NL : Initially the current default light scene is the default light scene for the case that the outdoor light sensors have no malfunction.

Property *I_M_R2*

Formal : $curLightScene = curDefaultLightScene$

NL : Initially the current light scene is the current standard light scene.

Figure 9: Description Class *Room*

Precision by Product Model

The product model forces a developer to distinguish between the relevant phenomena and to introduce a corresponding name for each phenomenon. In the case of the different kinds of light scenes mentioned in the original problem description, this activity results in the introduction of the four timed functions *curLightScene*, *curDefaultLightScene*, *malDefaultLightScene*, and *noMalDefaultLightScene* (see [Fig. 9]).

Completeness by Process Model

The process model includes a guideline that an *initial value* or the range of possible initial values should be specified for each machine controlled entity. Note that it is important that customer and developer also clarify the initialization of machine controlled entities as early as possible in the development process. In the case of the description class *Room*, this guideline yields the two properties *I_M_R1* and *I_M_R2*. This information is not provided by the original problem description, but has been acquired by interaction with the customer.

Reuse by Object Redeclaration

Description class *Room* includes a further kind of specialization, a so-called *object redeclaration* of the object *light* that *Room* inherits from description class *Area*. Due to the declaration

Object *Area::light* : *DimmableCeilingLightGroup*

the ceiling light groups in a room are not only objects of the description class *CeilingLightGroup*, but of its specialization *DimmableCeilingLightGroup*.

4.3.3 Office and OfficeNeighbor

Based on the description class *Room*, we specified *OfficeNeighbor*, *OfficeWithoutNeighbor*, *NoOfficeNeighbor*, and *NoOfficeWithoutNeighbor*. These four different specializations depend on whether a room is an office or not and whether a room has doors to neighbor rooms or not. In [Fig. 10] and [Fig. 11], we present one of these four "paths".

Expressiveness by Natural Language

According to the original problem description, the only significant difference between offices and non-offices is expressed in user needs U11 and U12 that deal with the installation of the control panels. This is also reflected in our description classes *Office* and *NoOffice*. They only differ in the *nonformal* property *R_O1* (see [Fig. 10]) and property *R_NO1* (see [KP00]). These properties as well as the non-functional properties NF6 - NF8 of [PD00] cannot be formalized within our approach. To keep a FOREST problem specification complete w.r.t. the original problem description, such *nonformal properties* may be entirely expressed in natural language.

Description Class Office

⋮

BASE CLASSES
Class *Room* ...
Intention : All definitions of description class *Room* are inherited.

REQUIREMENT SPECIFICATION
Property *R_O1*
NonFormal : The control panel should be movable in the offices like a telephone with a cord.

Figure 10: Description Class *Office*

Description Class OfficeNeighbor

⋮

BASE CLASSES
Class *Office* ...
Intention : All definitions of description class *Office* are inherited.
Class *NeighborRoomConnections* ...
Intention : All definitions of description class *NeighborRoomConnections* are inherited.

Figure 11: Description Class *OfficeNeighbor*

4.3.4 HallwaySection

Similar to the description class *Room*, the description class *HallwaySection* is derived from the description class *Area* (details in [KP00]).

4.4 Main Description Class Floor

The final description class in a problem specification is always the so-called *main description class*. In this description class, all objects belonging to a system are collected and related. In the LCCS-PS, the main description class is *Floor* [Fig. 12]. Here, we aggregate all rooms, hallway sections, staircases, and outdoor light sensors. Furthermore, we specify relations between these objects that can only be stated on this top level. For example, property *D_F1* specifies that the rooms *o435* and *o433* share a door. Using such explicite equalizations, we model the architecture of the floor as depicted in Figure 1 of the original problem description.

Reuse by Object Domains

Finally, we point out *object domains* which is a further concept of specialization. The domain *ROOMS* introduced in the signature of description class *Floor* is

Description Class Floor

SIGNATURE

Object $o435$: $OfficeNeighbor($ $numDoorHallway = 1$, $numDoorNeighbor = 1$ $)$

Intention : This object represents the office with room number 435, which is equipped with
- one door to the hallway
- one door to a neighbor room

Object $o433$: $OfficeNeighbor($ $numDoorHallway = 1$, $numDoorNeighbor = 2$ $)$

Intention : This object represents the office with room number 433, which is equipped with
- one door to the hallway
- two doors to neighbor rooms

Object $o431$: $OfficeNeighbor($ $numDoorHallway = 1$, $numDoorNeighbor = 1$ $)$

Intention : This object represents the office with room number 431. This office is equipped with
- one door to the hallway
- one door to a neighbor room

⋮

Domain $ROOMS = \{o435,\ o433,\ o431,\ \ldots\ , hl410\} \subseteq Room$

Intention : This domain contains all rooms of the specified floor.

⋮

DOMAIN KNOWLEDGE

Property D_F1

Formal : $\Box(o435.neighborDoor[1] = o433.neighborDoor[1])$

NL : The two rooms with room numbers 435 and 433 are connected by a door.

Property D_F2

Formal : $\Box(o433.neighborDoor[2] = o431.neighborDoor[1])$

NL : The two rooms with room numbers 433 and 431 are connected by a door.

⋮

MACHINE SPECIFICATION

Property M_F2

Formal : $\Box(\forall r \in ROOMS : (r.OLSMalFct \rightarrow r.curDefaultLightScene = r.malDefaultLightScene))$

NL : Whenever the machine assumes that some outdoor light sensor used by a room is not working correctly, then the default light scene is the default fault light scene, i.e. all ceiling lights are on.

⋮

Figure 12: Description Class *Floor*

an example of such an object domain. In an object domain, a developer can collect objects of the same description class that have already been declared. For this collection, which is a subset of a description class, a developer can specify additional properties. Property *M_F2* of description class *Floor* is an example of such an additional property. This is our formalization of the non-functional need NF_2 of the original problem description [PD00].

5 Discussion

In this section, we analyze the problem specification of the *Light Control Case Study* and discuss the strengths and shortcomings of the Forest approach.

5.1 Analysis of the LCCS-PS

In the LCCS-PS, we specified 44 description classes in eight groups. The final Postscript$^{\mathrm{TM}}$ document without prereferences to the original problem specification and without postreferences to the properties where a signature entry is used has 112 pages, with these references, it has 139 pages.

In the 44 description classes, we have introduced 162 names for phenomena/objects and 174 names for properties *explicitly*. *Implicitly*, we consider over 10000 phenomena and over 6000 properties. *Explicitly* means that a name occurs in a Signature part or in one of the three specifications parts Domain Knowledge, Requirement Specification, Machine Specification. The *implicitly* specified number of names is determined by expansion of the objects aggregated in the main description class *Floor*. For example, in the description class *Room*, the timed function name *curLightScene* is explicitly specified only once. But due to the aggregation of 20 different rooms in description class *Floor*, this function name is implicitly specified 20 times.

These numbers make it evident that the specified LCCS deals with a large system. On the other hand, they reveal the degree of compactness that is achieved by the structuring concepts of the Forest approach. Without such structuring concepts, the handling of large systems is practically impossible.

In the following, we provide some data concerning the kind of formulae in the LCCS-PS: 167 of the 174 properties obtained from the problem description have been formalized, i.e., they were specified as *formal properties*. 33 properties were concerned with the initialization of entities (as *I_M_R1* and *I_M_R2* in [Fig. 9]). They could be formalized in classical first order logic, i.e. without any temporal operator. From the remaining 134 properties containing at least one temporal operator, 125 properties could be formulated by applying only five(!) different requirement patterns. [Table 1] shows how often each of these patterns was instantiated. Additionally, the table contains the syntactical part of the pattern definition, which appears in the *Formal* part of the involved properties.

Regarding the simple invariance requirement pattern, we emphasize that in $\Box\varphi$, the formula φ does not contain any temporal operator. Hence, φ is in general a first order formula. In 87 instantiations of the invariance requirement pattern, the formula φ even does not contain any quantifier.

#Inst.	Pattern Name	Syntactical Part	Comment
8	*DelayedImplication*	$\Box(\varphi \Rightarrow_{\leq\tau} \psi)$	$\varphi \Rightarrow_{\leq\tau} \psi$ means that if φ is valid for at least τ time units, ψ becomes valid within the τ time units and remains valid at least as long as φ.
5	*DelayedCopy*	$\Box(s_1 \rhd^f_\tau s_2)$	$s_1 \rhd^f_\tau s_2$ means that s_2 has a value that is the result of function f applied to a value that s_1 has had sometime during the last τ time units.
8	*BoundedResponse*	$\Box(\varphi \rightarrow \Diamond_{\leq\tau}\psi)$	$\Diamond_{\leq\tau}\psi$ means that ψ is true within the next τ time units.
3	*LimitedInvariance*	$\Box([\varphi] \rightarrow (\Box_{\leq\tau}\varphi))$	$[\varphi]$ means that φ becomes true at the current time point. $\Box_{\leq\tau}\varphi$ means that φ is true for the next τ time units.
101	*Invariance*	$\Box\varphi$	$\Box\varphi$ means that φ is always true.

Table 1: Requirement Pattern Instantiations

In the end, most properties could be formalized using a small number of requirement patterns – only nine temporal formulae had to be formalized conventionally. This shows that in spite of the decision for a formal description technique, the knowledge of a few typical patterns is sufficient to understand a large portion of the formal part of the problem specification.

5.2 Strengths and Shortcomings of the FOREST approach

To complete the results from the application of the FOREST approach to the LCCS, we will now reconsider the objectives of [Section 2] and discuss to what extent these objectives are achieved by the FOREST approach.

Precision by Logic, (Requirement) Patterns, and Reference Model Precision is achieved by the usage of a formal description technique (FDT). 96 percent of the properties obtained from the problem description have been formalized. The remaining properties are rather vague, for instance, NF8: ,,*The control panels are easy and intuitive to use*", and are therefore difficult to formalize in general.

Where applicable, requirement patterns clearly help to avoid mistakes and to complete specifications. Nevertheless, there is still a risk of over-specification, if a pattern is instantiated at a place where it is too restrictive. In the end, this decision is left to the requirements engineer.

Intelligibility by Structuring, Natural Language, and Patterns

If the specification structure is correlated with some common structure of the real world (e.g., the building structure), this facilitates understanding. Unfortunately, information is often scattered over several description classes. For example, the information of an office with connections to neighbor rooms (*OfficeNeighbor*) is contained in *Office*, *NeighborRoomConnections*, *Room*, and *Area* (see [Fig. 7]). Generally, aggregation and inheritance increase the intelligibility by creating new levels of abstraction. On the other hand, it can be laborious to collect all information relevant for a description class by traversing all its base classes.

The explanation of each formal specification part in natural language is very useful, but can be potentially inaccurate. Together with the temporal logic that was chosen to support a property-oriented specification style, the resulting translations into natural language are close to the problem description given by the customer.

Patterns lead to a kind of normalization of either properties and corresponding translations into natural language.

Expressiveness by Logic and Natural Language

The expressiveness of the FOREST approach is mainly determined by the decision for a real-time temporal logic. Natural language must be used, where necessary (see above).

Scalability by Structuring

The object-oriented structuring concepts increase the scalability in the sense that they support the treatment of *large* systems.

Reuse by Structuring and Patterns

The reusability of specification artifacts obviously depends on their universality and on the application domain in which they shall be reused. In the LCCS, it was possible to take parts from former case studies (see `http://rn.informatik.uni-kl.de/~forest/examples/`). These were, for example, the overall structure given by the groups, since it is typical for this kind of reactive systems, complete description classes, especially on lower specification levels, such as the *NoMalfunctionSensor* in [Fig. 2], and requirement patterns on the level of properties.

Traceability by References and Tool Support

The traceability of dependences between the original problem description and the formal problem specification, as well as within the formal problem specification, is improved by references from and to the related specification parts. A well defined document structure is a precondition for expressive references and the corresponding tool support.

Practicability by Tool Support

The tool *xforest* currently supports the creation of syntactically correct problem specifications and manages the traceability relations. It can produce different projections of the specification, especially in HTML format, where the traceability relations are realized by hyper links. Without *xforest*, it would have been difficult to create the LCCS-PS!

Most of the discussed shortcomings could be improved by suitable tool support. For example, a *semantic analysis* option would increase precision. *Graphical representations*, additional *projections*, and the possibility to generate an *executable prototype* could further increase intelligibility (this would probably require the addition of an operational description technique).

6 Conclusion

The *Light Control Case Study* is an example of a non-trivial reactive system. We have shown that the FOREST approach is applicable to this case study, and that the structuring concepts of FOREST have a positive impact on a range of typical specification objectives, namely scalability, reuse, and intelligibility. Requirement patterns could be successfully applied in many cases. We have confirmed that the distinction between machine and environment as required by all of the mentioned reference models, is substantial for the elaboration of the problem, and that the applicability and acceptance of a formal specification technique strongly depends on the availability of a suitable tool.

Acknowledgements

We thank Reinhard Gotzhein for his valuable comments on this paper and the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 501, "Development of Large Systems with Generic Methods" for supporting this work.

References

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 1999.

[FBWK92] S. Faulk, J. Brackett, P. Ward, and J. Kirby. *The core method for real-time requirements.* IEEE Software, 9(6):22–33, September 1992.

[GGJZ00] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave. *A reference model for requirements and specifications.* IEEE Software, 17(3):37–43, May/June 2000.

[GKP98] R. Gotzhein, M. Kronenburg, and C. Peper. *Reuse in requirements engineering: Discovery and application of a real-time requirement pattern.* In A. P. Ravn and H. Richel, editors, Proc. of the 5th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, Lyngby, Denmark, pages 65–74. Springer, 1998.

[KP99] M. Kronenburg and C. Peper. *Definition and instantiation of a reference model for problem specifications.* In 11th International Conference on Software Engineering and Knowledge Engineering (SEKE'99), pages 332–336, 1999.

[KP00] Martin Kronenburg and Christian Peper. *Problem specification of* The Light Control Case Study *of the Journal of Universal Computer Science.* http://www-avenhaus.informatik.uni-kl.de/forest/EXAMPLES/JUCS/JUCSStart.html, 2000.

[KPG96] M. Kronenburg, C. Peper, and R. Gotzhein. *A tailored real time temporal logic for specifying requirements of building automation systems.* SFB 501 Report 16/96, University of Kaiserslautern, 1996.

[PD00] *The Light Control Case Study: Problem Description.* Journal of Universal Computer Science, Special Issue on Requirements Engineering (This volume), 2000.

[PGK97] C. Peper, R. Gotzhein, and M. Kronenburg. *A generic approach to the formal specification of requirements.* In Proc. of the 1st IEEE Intl. Conf. on Formal Engineering Methods (ICFEM'97), Hiroshima, Japan, pages 252–261, 1997.

[PM95] D. L. Parnas and J. Madey. *Functional documentation for computer systems.* Science of Computer Programming, 25:41–61, 1995.

[SS99] T. Schmidt-Samoa. *FoReST – Entwurf und Implementierung einer Umgebung zur Erstellung formaller System-Anforderungsbeschreibungen* (in German). University of Kaiserslautern, Computer Science Departement, Diploma Thesis, October 1999.

[vSPM93] A. J. van Schouwen, D. L. Parnas, and J. Madey. *Documentation of requirements for computer systems.* In Proc. of IEEE International Symposium on Requirements Engineering, pages 198–207. IEEE Computer Society Press, 1993.

[ZJ97] P. Zave and M. Jackson. *Four dark corners of requirements engineering.* ACM Transactions on Software Engineering and Methodology, 6(1):1–30, 1997.