Journal of Universal Computer Science, vol. 6, no. 7 (2000), 597-620
submitted: 3/1/00, accepted: 20/4/00, appeared: 28/7/00

Capturing Requirements by Abstract State Machines: The Light Control Case Study

Egon Börger
(Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it – Visiting Microsoft Research, Redmond)

Elvinia Riccobene
(Università di Catania, Dipartimento di Matematica e Informatica,
I-95125 Catania, Italy
riccobene@dmi.unict.it)

Joachim Schmid
(Siemens AG, Corporate Technology, D-81730 Munich, Germany
joachim.schmid@mchp.siemens.de)

Abstract: We show how to capture informally stated requirements by an ASM (Abstract State Machine) model. The model removes the inconsistencies, ambiguities and incomplete parts in the informal description without adding details which belong to the subsequent software design. Such models are formulated using application-domain-oriented terminology and standard software engineering notation and bridge the gap between the application-domain and the system design views of the underlying problem in a reliable and practical way, avoiding any formal overhead. The basic model architecture reflects the three main system parts, namely for the manual and automatic light control and for handling failures and services. We refine the ground model into a version that is executable by AsmGofer and can be used for high-level simulation, test and debugging purposes.

Key Words: Requirements, ASM, Light Control, Case Study, Refinement.

Category: D.2.1

1 Introduction

Despite intensive research in the area of requirements engineering there is no agreement either on the process to follow or on the methods to use, for going from an informally stated software engineering problem to a formulation which can be integrated into the subsequent design process and therefore has to be complete, consistent, abstract but rigorous enough to serve as faithful reference for the design. We illustrate in this paper an approach to requirements capture, which through analysis turns informally stated requirements into a rigorous requirements specification providing the possibility

- to check by analytical means the internal consistency and the intrinsic completeness of the requirements, in terms of the rigorous specification,

– to analytically and experimentally check the correctness and completeness of the rigorous specification with respect to the informal requirements (faithfulness and adequacy),

– to adapt the specification to requirement changes which occur during the design,

– to formulate an unambiguous "contract" between the application domain expert, the "customer", and the system designer. This contract represents for the customer the binding development goal and for the system designer a reliable, i.e. clear, stable and complete, starting point for the implementation.

These properties demand that the requirements specification represents a functionally complete but abstract description of sufficient but not more than necessary rigor which

1. can be read and understood by and justified to the customer as corresponding to what he wants,

2. defines every system feature as far as this is semantically relevant for the work the user expects the system to achieve (avoiding underspecification),

3. contains only what the logic of the problem requires for the system behavior (avoiding overspecification), i.e. does not rely upon any further design decision belonging to the system implementation (like the representation of objects, the serialization of parallel actions).

Such high-level specifications of requirements – so-called *ground models* [Börger 1999] – have to solve pragmatically the purely theoretically unsolvable problem to link in a "justifiably correct" way real-world problems to machine models, i.e. vague largely natural language descriptions to formal code which is governed by mathematical laws. They represent the authoritative formulation of the requirements against which the implementation has to be checked and tested. Using Abstract State Machines (ASMs), one can provide the needed conceptual, experimental and mathematical justification for abstract models of given requirements. Indeed, on one hand, ASM models can be tailored to the (abstraction level of the) application domain problem and thus be analysed and compared to the real-world situation by direct inspection. On the other hand, the rigorous nature of ASMs allows one to formulate conditions for system validation (test) and verification (proof), for objective and repeatable machine or thought experiments (elaboration of test plans), together with internal consistency checks. In this paper, we illustrate this technique by turning the requirements given for the Light Control Problem [Problem Description 2000] into an ASM which constitutes an executable ground model, a satisfactory starting point for the proper software design.

1.1 The process of ground model construction

For building a ground model there are three things we have to do. The first is to *collect* the informally presented requirements information and to disambiguate it, removing the unintended ambiguities and producing a sufficiently precise formulation (elicitation). The unambiguous description extracts from the informal requirements the involved basic domains (types of objects), the appearing basic operations and the basic relations among the objects and the domains, in a word the "object-oriented" structure of the system. This part of the resulting description is often called model signature (and indeed will constitute the signature of the ASM we are going to develop for the Light Control Problem). During the following design, the signature is typically transformed into data structures, header and interface definitions. The semantic link of the relevant high-level terms to the application domain notions and to the basic operation sequences is typically documented in a lexicon or through use cases (user scenarios). Through checklists which relate informal to formal terms and vice versa, for example by using hypertext links, one has a) to document that nothing has been forgotten and b) to guarantee that the formalization is traceable. Through the rather Socratic method of asking ignorant questions [Berry 1995] one can try to make sure that the semantic interpretation of the informal problem description is correctly captured in the mapping to the terms of the rigorous description.

The second thing to do is to *structure* the resulting description to make it more transparent and more easily checked for internal consistency. Structuring prepares the description for change by parameterization and abstraction. It also reveals similarities and commonalities among the requirements. During this structuring work, it is crucial not to loose traceability to the informal requirements. This structuring work has also to make explicit the basic architectural features which are implicitly imposed by a possible solution for the given problem under the given constraints.

The third step is to *complete* the resulting description by exhibiting and filling in, typically through additional requirements coming from the customer, all the information which is missing in the informal problem description but is necessary for a full problem statement. Typically this includes the analysis of boundary conditions, of exception handling, of robustness features, etc. This activity eventually turns the description into a ground model of the system to be developed. This high-level model incorporates all the requirement (not design) decisions which are relevant from the point of view of the application domain (customer system view) and have to be documented explicitly during the process of building the ground model. It helps checking and establishing the completeness and correctness of ground models that ASM models are machines, easy to understand for the practitioner, which can also be made executable so that high-level features of the system to be developed – in particular scenarios

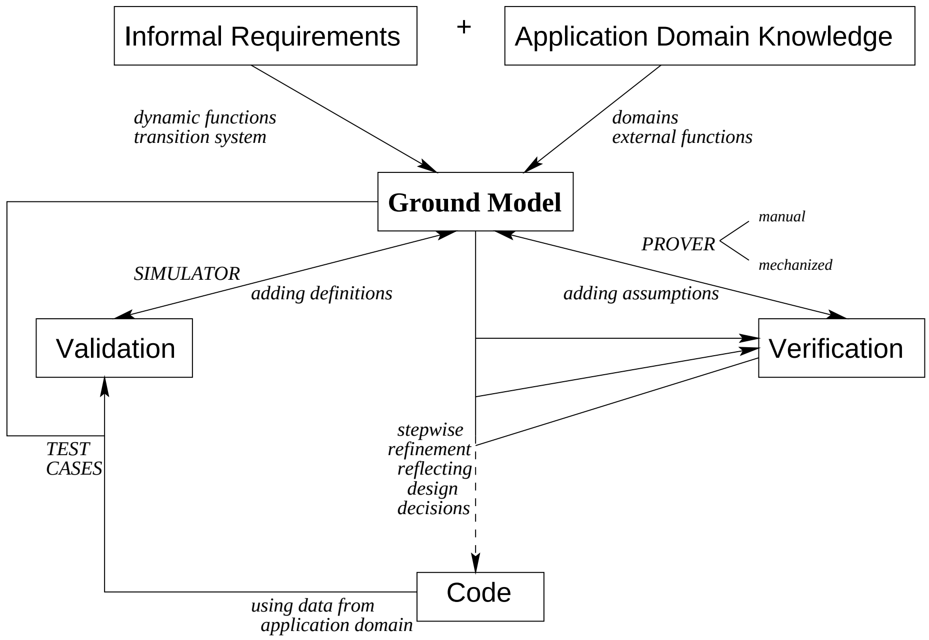

Figure 1: Development Process

– can be simulated and visualized by the customer.

This process of ground model construction is usually not linear, but iterative, and typically is accompanied by extensive simulation, high-level proving activities, layout of test plans, etc. (see [Figure 1]). Nevertheless, it is important to document not only the executable code but also the *final* ground model – which in the worst case will be completely defined through later changes which show up only during implementation – to guarantee the traceability of the initial requirements to the code and to enhance the maintainability and extendability of the resulting system. This implies in particular that during the design process, the ground model is maintained and kept to correspond to the actual design. Since ground models "are easier to read than the programs that they describe" [Parnas 1999, p.195], they support the software developers' daily work and improve its quality as advocated by Parnas.

1.2 Application to the Light Control Problem

The following sections illustrate for the Light Control Problem this requirements engineering process by explicitly defining the signature, the structure, and the complete list of additional requirement decisions which together result in the construction of an ASM ground model for the Light Control Problem. We have made this ASM executable using AsmGofer [Schmid 1999a], essentially by providing definitions of the features which remain abstract in the ground model, so that the user can do experiments with the requirements (run user scenarios and test suites).

We will concentrate our attention here on the ground model construction as a method to disambiguate informally presented requirements, to structure them, to analyse them (with respect to internal consistency and correctness) and to complete them, from the customer's, not the designer's point of view, in a way which makes them prototypically executable. As a consequence in this paper we do not investigate the layout of test plans from the ground model, and we also do not investigate any properties one might wish to be proved for the ground model. Furthermore we do not adhere to any systematic documentation or traceability practice, as is mandatory for industrial applications of the method. In particular, we do not document here the iterative process of ground model construction (which includes our reaction to the customer feedback [Customer Feedback 1999]), but only its final outcome.

We do however list explicitly all those additional requirements which we introduce along the way in order to disambiguate the given informal requirements, to make them consistent and reasonably complete. This requirements completion is a crucial part of capturing requirements and in practice has to be worked out in close cooperation with the customer. The present problem description [Problem Description 2000], despite repeated revisions due to customer feedback [Customer Feedback 1999] reported in the introductory note, still contains numerous inconsistencies, incoherences and ambiguities which showed up during the ground model construction. This fact is typical and shows the necessity to build requirements ground models as a safe basis for software design. Here are some examples of problems we identified in the informal requirements: missing priority requirements to avoid inconsistencies between actions involving "shared actors" (in particular between manual and automatically triggered actions); the inconsistency of the informal requirements concerning the light scene upon reentering a room; the ambiguity of notions like room occupation or uncontrollability of hallway light; the incomplete definition of various basic concepts like switching off lightgroups, pushing buttons, selecting ambient light scenes, etc. The complete list appears in the appendix and is explained in the following sections. If our decisions for disambiguating the informal requirements do not always reflect the result of the customer feedback [Customer Feedback 1999], as one reviewer pointed out, this is because our model has been built and made executable for the Dagstuhl meeting [Börger et al. 1999a], long before that customer feedback became available. The interested reader is invited to experiment with adapting our model to different decisions, to experience how naturally ASM models can be extended or otherwise modified. For this very same reason of "design for change" we also pay attention to build our model in a parametric way, exploiting the abstraction features offered by ASMs.

2 Notation and Prerequisites

We use only standard notation and therefore invite the reader to skip this section and to come back to it only should the necessity arise. Our notation includes ASMs, which represent a semantically well-defined, precise form of widely used pseudo-code over abstract structures. We provide in the rest of this section some intuitive explanations which should suffice to correctly understand and use ASMs for turning informally stated requirements into a rigorous form. We refer the reader to [Gurevich 1995] for a detailed mathematical definition.

The states of ASMs are arbitrary structures in the standard sense they are used in mathematical sciences, i.e. domains of objects with functions and predicates defined on them. The basic operations of ASMs are guarded destructive assignments of values to functions at given arguments, expressed in the following form:

if *cond* **then** *Updates*

where *cond* is an arbitrary condition (boolean expression) formulated in the given signature, *Updates* consists of finitely many *function updates*:

$$f(t_1, \ldots, t_n) := t$$

which are executed simultaneously. The terms $t_1, \ldots, t_n$ are arguments at which the value of the arbitrary function f is set to t. For technical convenience we treat predicates as boolean-valued functions.

An ASM M is a finite set of rules for such guarded multiple function updates. The computation of an ASM is defined in the standard way transition system runs are defined. Applying one step of M to a state $\mathcal{A}$ produces as next state another state $\mathcal{A}'$, of the same signature, obtained as follows: First evaluate in $\mathcal{A}$, using the standard interpretation of classical logic, all the guards of all the rules of M. Then compute in $\mathcal{A}$, for each of the rules of M whose guard evaluates to true, all the arguments and all the values appearing in the updates of this rule. Finally replace, simultaneously for each rule and for all the locations in question, the previous $\mathcal{A}$-function value by the newly computed value if no two required updates contradict each other. The state $\mathcal{A}'$ thus obtained differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by a rule of M which could fire in $\mathcal{A}$. The effect of an ASM M, started in an arbitrary state $\mathcal{A}$, is to repeatedly apply one step of M as long as an M-rule can fire. Such a machine terminates only if no rule is applicable any more (and if the monitored functions do not change in the state where the guards of all the M-rules are false).

We freely use standard notational extensions like *case of*, *let* and *where*. We also use the additional ASM rule construct *forall* x *with* P *do* R with the intended meaning that rule R is executed simultaneously for all values of x which satisfy the property P. We also use macros, parameterized ASMs and a natural concept of submachines which are defined more precisely in [Börger and Schmid 2000]. In addition we make use of the ASM function classification which we are going to explain below.

It turned out to be practically useful to distinguish, in an ASM M, *basic* functions from *derived* functions (which are defined in terms of basic ones). Many requirements can be formalized in this way, by reducing them through definitions to basic terms. This simple technique, long established in mathematical sciences, provides a powerful high-level modularization and information hiding mechanism.

Within derived or basic functions, *static* functions, which remain constant during M-computations, are distinguished from *dynamic* ones, which may change from M-state to M-state. Many application domain features showing up in requirements are of a static nature and can be described independently of the dynamics of the system to be built. This separation of concerns helps enormously to keep high-level models of requirements small and transparent, graspable by the human reader, avoiding the rightly criticised [Parnas 1999] size explosion coming with most formalization methods in the literature.

Adapting the terminology introduced in Parnas' Four Variable Model [Parnas and Madey 1995] we distinguish among the dynamic functions the *controlled* ones from the *monitored* ones which we also call *in* functions. The controlled functions are subject to change by an update appearing in a rule of M. The *monitored* functions can change only due to the environment or, more generally, due to actions of other agents. Controlled functions can be read and written by M, they are functions f of M-updates $f(t_1, \ldots, t_n) := t$ and are allowed to appear also in the arguments and values t_i, t. The *in*-functions typically serve to read system *input* (dynamic interface event). The way they are updated depends on the specific system. The natural pendant to *in* functions are *out*-functions which can only be written by M, i.e., appear only as f in M-updates $f(t_1, \ldots, t_n) := t$ but nowhere else, in particular not in t_i, t and not in rule guards. Last but not least, there are *shared* functions which can be written and read by M and by some other agent and for whose consistency usually a protocol has to be devised. Shared functions help to naturally reflect multi-agent computations and combined read and write use of locations, like ports in chip design which are used for both input and output. This classification is pictorially represented in [Figure 2] and extends the classification appearing in

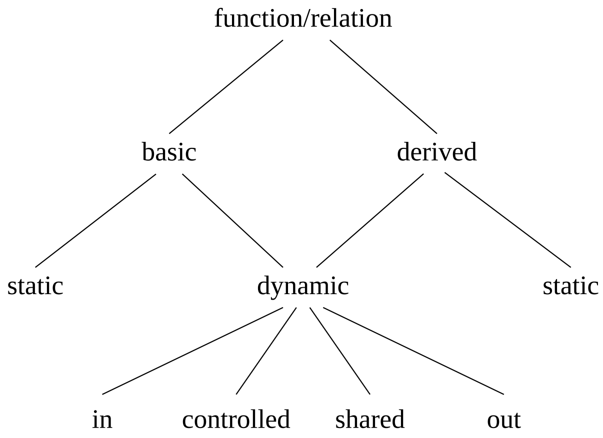

Figure 2: ASM Function Classification

[Börger 1999].

For the case study to be discussed here, we use a particular form of monitored functions, namely *events* which typically appear in the guards of rules and for which it is assumed that they are *consumed by firing the rule* (read: their value is reset to undef when the rule is applied). For the case study we deliberately leave the details of the event model open and limit ourselves to specify as contraints those conditions which are explicitly imposed by the requirements. In the following we need in particular the case of boolean-valued events which, by becoming true, enable a rule to be fired and become instantly false by firing the rule. The special feature of events is that if e enables at clock time (*computation time*) t a rule which is fired, its being consumed does not exclude that e may have an enabling value again at clock time $t+1$, namely in the case when, between the system's computation moments t and $t+1$, the event happens once again. These two instances represent different occurrences of the event.

For modularization purposes we use parameterized submachines in a way which supports established programming practice. For a precise definition of these concepts, in the context of parallel execution of multiple ASM rules, we refer the reader to [Börger and Schmid 2000].

3 Capturing the Requirements

In this section we analyse the given Light Control requirements and express them in a mathematical form. We introduce formal pendants (the signature) for the various objects and operations mentioned in the document and use definitions and algorithmic notation (ASM rules) to express their static and dynamic dependencies. To facilitate checking the correctness and the completeness of our modeling, we largely follow the order in which the requirements are presented in [Problem Description 2000]. This implies that we develop the model in a bottom up manner, although for a better understanding of the resulting entire model, a top down reading will be helpful.

After the short introduction, [Problem Description 2000] begins with the floor description (Part 2) to define the signature of the system to be developed, i.e. a listing of its components with the associated data structures. We reflect this signature definition by listing and classifying the basic classes of system objects and their properties. In Part 3, [Problem Description 2000] distinguishes three categories of informal needs, the user needs, the facility manager needs, and fault tolerance. We obtain a modularization of the requirements by parameterizing and grouping the user and the facility manager needs into the requirements for the possible manual interactions of the user or facility manager with the control system, such as pressing buttons (reflecting essentially Section 3.1 of [Problem Description 2000]), and into the automatic actions which are triggered by the control system, such as switching off light in unoccupied rooms. At the end we describe the malfunctions (reflecting Section 3.2 of [Problem Description 2000]).

This is already a good place to introduce the formal representation of a distinction which is made in [Problem Description 2000] for each category of needs, namely between actions which are triggered by the user or the facility manager and actions which are triggered by the control system (due to some calculation on the basis of external information). We express this distinction by a dynamic controlled function *mode* which indicates whether the current light setting operation in a room has been done manually, by the user or the facility manager, or automatically. In an object-oriented perspective this function can be thought of as being parameterized by rooms.

3.1 Basic objects and operations (Signature)

The basic objects appearing in the floor description are rooms and hallways, more precisely hallway sections. They are associated with light groups (window and wall ceiling lights for rooms and ceiling lights for hallways) which come with operations of pushing various buttons (on the wall or a control panel) and of actuating dimmers. Rooms and hallways are also associated with various motion detectors, light sensors, and door closing contacts. There are also status lines which report status values of the associated light groups.

The staircases, which are mentioned in the Floor Description, enter the problem really only through their motion detector. To avoid the proliferation of irrelevant object types we include these staircase motion detectors in the class of motion detectors which are related to the doors for entering a hallway from a staircase.

These objects enter our formalization as parameters of the various actions which are described in the following sections. In a systematic documentation of the requirements elicitation one has to list explicitly the complete signature; the basic objects have to be defined through the lexicon, their properties as well as the conditions imposed on the operations have to be listed systematically, making sure that the list is complete and correct with respect to the application domain information which underlies [Problem Description 2000]. We abstain from doing this here; any reliable, systematic method can serve this purpose.

3.2 User interactions (Manual light control)

[Problem Description 2000] states in Section 2.1-2.4, Paragraphs 7-11 that every room has two wall switches, one for the wall ceiling light group and the other for the window ceiling light group. The behavior, if the corresponding push button is pushed, is formulated in Section 2.1.2 as follows.

1. If the ceiling light group is completely on, it will be switched off.
2. Otherwise it will be switched on completely.

We capture this requirement with the rule *Room_wall_button*. This rule is parameterized over a room and a light group. The fact that pressing the button is an external event over which the system has no control is reflected by an event function *lightgroup_wall_button_pressed* which is supposed to become true when the corresponding button has been pressed and to become false when the rule fires whose guard contains the event function (PushButtonReq)[1]. This interpretation resolves the incompleteness of the definition for *push button* in the dictionary where it is not made clear when the light effect should take place, at the beginning or at the end of the possibly prolonged button pushing action.

$$
\begin{array}{l}
\mathit{Room\_wall\_button}(\mathrm{room}, \mathrm{lightgroup}) = \\
\quad \textbf{if } \mathit{lightgroup\_wall\_button\_pressed}(\mathrm{room}, \mathrm{lightgroup}) \textbf{ then} \\
\quad\quad \textbf{if } \mathit{lightgroup\_is\_completely\_on}(\mathrm{room}, \mathrm{lightgroup}) \textbf{ then} \\
\quad\quad\quad \mathit{Switch\_lightgroup\_off}(\mathrm{room}, \mathrm{lightgroup}) \\
\quad\quad \textbf{else} \\
\quad\quad\quad \mathit{Switch\_lightgroup\_completely\_on}(\mathrm{room}, \mathrm{lightgroup})
\end{array}
$$

Notation: We use *italic* font for rules and functions. Rule names always start with an initial capital letter and function names with a lower case letter. Local variables are denoted in roman font and we use `typewriter` for constants.

[1] Such additional requirements are provided for the purpose of a systematic documentation of all the decisions taken to interpret or extend the requirements in [Problem Description 2000].

Switching a light group off and switching it completely on is defined as setting all lights in the corresponding room to `minDimValue` or `maxDimValue` respectively [2]. This definition resolves the apparent contradiction in U1, considering it as safe to allow a person who wants to rest in a room to choose a light scene in which all the lights are switched off and the room is dark (`U1`Req).

As explained above, *mode* determines for each room whether the light was set by the user (`Manual`) or by the control system (`Ambient`)[3].

$Switch\_lightgroup\_off(\mathrm{room}, \mathrm{lightgroup}) =$
  $mode(\mathrm{room}) :=$ `Manual`
  forall light $\in$ $lights\_in\_group(\mathrm{room}, \mathrm{lightgroup})$
    $Switch\_light(\mathrm{room}, \mathrm{light}, \mathtt{minDimValue})$

$Switch\_lightgroup\_completely\_on(\mathrm{room}, \mathrm{lightgroup}) =$
  $mode(\mathrm{room}) :=$ `Manual`
  forall light $\in$ $lights\_in\_group(\mathrm{room}, \mathrm{lightgroup})$
    $Switch\_light(\mathrm{room}, \mathrm{light}, \mathtt{maxDimValue})$

Following Section 2.6, in every hallway section there are switch buttons, linked in parallel (see 2.6.3). The light in the hallway section has to be on if one button is defective (*any_hallway_button_defect*). This is according to `NF5` where we interpret "not controllable manually" in view of the safety requirement `U1`, as meaning that at least one hallway button is defective (`NF5a`Req). It would be easy in our ASM model to change this to a "local" interpretation of "not controllable manually" in `NF5` by parameterizing *any_hallway_button_defective* by buttons as arguments. The event function *hallway_button_pressed* indicates that a switch button has been pressed and becomes false by firing the rules in which the event appears in the guard.

$Hallway\_button(\mathrm{hallway}) =$
  if $hallway\_button\_pressed(\mathrm{hallway}) \wedge$
    $\neg any\_hallway\_button\_defect(\mathrm{hallway})$ **then**
    if $light\_is\_on(\mathrm{hallway})$ **then**
      $Switch\_lights\_off(\mathrm{hallway})$
    else
      $Switch\_lights\_on(\mathrm{hallway})$

According to the dictionary definition of "push button", switching light on or off manually in rooms and hallways is similar. We reflect this uniformity by

[2] Following good system development practice we use symbolic names rather than constants, like 0% and 100%.

[3] This interpretation of `Ambient` does not preclude to let the light from the sun be part of what is understood by the environmental light.

introducing the term *location* standing for rooms and hallways. Switching on or off for a location is defined as setting all lights of the location to `minDimValue` or `maxDimValue` respectively. We group these lights for short as $lights\_at(location)$. Although for uniformity reasons we formulate *Switch_lights_on* for locations, we will use *Switch_lights_on* only for hallways, because by requirement `U5,U6,U9` and the dictionary entry "light scene", the light in a room is switched on only for a lightgroup as a whole.

$$
\begin{array}{l}
\mathit{Switch\_lights\_on}(\text{location}) = \\
\quad \textbf{forall } \text{light} \in \mathit{lights\_at}(\text{location}) \\
\quad\quad \mathit{Switch\_light}(\text{location}, \text{light}, \texttt{maxDimValue})
\end{array}
$$

$$
\begin{array}{l}
\mathit{Switch\_lights\_off}(\text{location}) = \\
\quad \textbf{forall } \text{light} \in \mathit{lights\_at}(\text{location}) \\
\quad\quad \mathit{Switch\_light}(\text{location}, \text{light}, \texttt{minDimValue}) \\
\quad \textbf{if } \mathit{location\_is\_room}(\text{location}) \textbf{ then} \\
\quad\quad \mathit{mode}(\text{location}) := \texttt{Manual}
\end{array}
$$

The facility manager can switch off the ceiling light in a room or hallway section if the room or hallway section is not occupied (`FM6`).

$$
\begin{array}{l}
\mathit{Manually\_switch\_off}(\text{location}) = \\
\quad \textbf{if } \mathit{manually\_switch\_off\_pressed}(\text{location}) \wedge \neg \mathit{occupied}(\text{location}) \textbf{ then} \\
\quad\quad \mathit{Switch\_lights\_off}(\text{location})
\end{array}
$$

The rule uses the function *occupied*. [Problem Description 2000] does not describe how to determine occupation. Rooms and hallways have motion sensors but these can sense only motion. Imagine somebody is sitting for a while quietly on his chair in a room so that the motion sensor reports no motion. However, the room is still occupied. Therefore a reasonable definition for a location to be not occupied is that there has been no motion for a period of *max_quiet_time* (RoomOccupationReq). It remains to be determined whether this function is fixed once and for all or whether it can be changed and what is its concrete value for a given system.[4] To reflect the malfunction requirement `NF4` we include the case that a location is occupied if at least one of its motion detectors does not work correctly, so that in this case the light cannot be switched off by the facility manager. To guarantee the consistency between user and facility manager light updates in the rules *Room_wall_button*, *Hallway_button*, *Manually_switch_off*, *Control_panel* we assume that the motion sensor detects when users push buttons (MotionDetectorReq). This is a semantic constraint which relates the notion of being occupied to the event functions *pressed* associated to buttons.

[4] Dan Berry remarked correctly that this definition does not take into account the case of somebody taking a nap who would not like to be disturbed by the light coming back due to moves in the sleep.

$occupied(\text{location}) =$
$\quad current\_time - last\_motion(\text{location}) \leq max\_quiet\_time$

The dynamic function *last_motion* stores the time of the last motion and we update the function by observing the motion detector. The monitored function *somebody_is_moving* yields the value of its location's motion detector. To satisfy `NF5`, we assume that the function *somebody_is_moving* is true if the corresponding motion detector is defect (`NF5b`Req).

$Observe\_motion\_detector(\text{location}) =$
$\quad \textbf{if } somebody\_is\_moving(\text{location}) \textbf{ then}$
$\quad\quad last\_motion(\text{location}) := current\_time$

There is one more action which is triggered by user interaction. According to `U5,U6,U9`, one can use the control panel to control the ceiling lights and the light scene. The ceiling lights can be switched on and off. Before collecting the requirements for the light scene we first analyse the overall functionality of the control panel. We use an additional event function *switch_value* to express the on or off position chosen by the user for the switch in question.

$Control\_panel(\text{room}, \text{switch}) =$
$\quad \textbf{if } switch\_pressed(\text{room}, \text{switch}) \textbf{ then case } \text{switch } \textbf{of}$
$\quad\quad \texttt{AmbientSelection} \rightarrow$
$\quad\quad\quad Activate\_light\_scene(\text{room}, last\_light\_scene(\text{room}))$
$\quad\quad \texttt{LightGroup}(\text{lg}) \rightarrow$
$\quad\quad\quad \textbf{case } switch\_value(\text{room}, \text{switch}) \textbf{ of}$
$\quad\quad\quad\quad \texttt{On} \rightarrow Switch\_lightgroup\_completely\_on(\text{room}, \text{lg})$
$\quad\quad\quad\quad \texttt{Off} \rightarrow Switch\_lightgroup\_off(\text{room}, \text{lg})$
$\quad\quad \texttt{SceneSelection} \rightarrow$
$\quad\quad\quad \textbf{case } switch\_value(\text{room}, \text{switch}) \textbf{ of}$
$\quad\quad\quad\quad \texttt{Scene}(\text{s}) \rightarrow Set\_light\_scene(\text{room}, \text{s})$

One has to guarantee that simultaneous pushing on wall buttons and on the control panel does not produce effects which exclude each other. One can for example assume that the hardware solves this conflict, or one could establish a fixed priority (PushButtonReq).

The preceding definition fulfills the informal needs `U5`, `U6`, and `U9`. The button `AmbientSelection` activates the light scene which was set by the action `SceneSelection` using the control panel.

$Set\_light\_scene(\text{room}, \text{scene}) =$
$\quad \textbf{if } mode(\text{room}) = \texttt{Ambient} \textbf{ then}$
$\quad\quad Activate\_light\_scene(\text{room}, \text{scene})$
$\quad \textbf{else}$
$\quad\quad last\_light\_scene(\text{room}) := \text{scene}$

The rule *Set_light_scene* for scene selection in ambient mode changes the current activated light scene and in manual mode simply stores the selected value in the dynamic function *last_light_scene*. In the second case the scene can be activated by pressing `AmbientSelection`.

A light scene contains an *ambient light level* and an ordered list of lights together with a dim value for each light (see Requirement 2.10, Paragraph 19). As the dictionary indicates under the entry light scene, the control system has to switch on the lights in the given order with the corresponding dim value in order to achieve the specified ambient light level. Reflecting `FM1` the control system must also take into account the ambient light from outside. We capture these requirements by introducing a function *lights_to_turn_on* which computes an ordered set containing all lights that should be switched on in this order together with their dim values (LightSceneReq). Introducing an order makes the dictionary definition of "light scene" uniform with respect to the way light scenes and their light groups are built from components, achieving easy adaptability to changing requirements. The function depends for each room on the value of the outdoor light sensor and of the activated light scene. This specification leaves still much freedom for detailing the structure of light scenes. The fact that we use this function only for rooms and not for hallways reflects that the requirements `FM1` and `NF3` are useless for hallways if they have no windows, as is suggested by Figure 1 in Paragraph 5 of [Problem Description 2000] (HallwayReq)[5]. Note that the derived function *lights_to_turn_on* (see below) takes into account the information about malfunctioning lights.

$Activate\_light\_scene(\mathrm{room}, \mathrm{scene}) =$
  $mode(\mathrm{room}) :=$ `Ambient`
  $last\_light\_scene(\mathrm{room}) := \mathrm{scene}$

  if $\mathrm{scene} = default\_light\_scene(\mathrm{room}) \wedge$
    $outdoor\_light\_sensor\_defect(\mathrm{room})$ **then**
      $Switch\_lights\_on(\mathrm{room})$
  else
    let $lights\_on = lights\_to\_turn\_on(\mathrm{room}, outdoor\_sensor(\mathrm{room}), \mathrm{scene})$
    forall $(\mathrm{light}, \mathrm{value}) \in lights\_on$
      $Switch\_light(\mathrm{room}, \mathrm{light}, \mathrm{value})$
    forall $\mathrm{light} \in lights\_at(\mathrm{room}) \setminus \{\mathrm{l} \mid (\mathrm{l}, \mathrm{v}) \in lights\_on\}$
      $Switch\_light(\mathrm{room}, \mathrm{light}, \mathtt{minDimValue})$

This rule also correctly reflects the requirement `NF2`:

[5] `FM1` and `NF3` are useful however if, as one reviewer remarked, light should be taken into account which may reach hallways through open doors of rooms. It is straightforward to adapt our model to take also such an interpretation of the requirements into account.

If any outdoor light sensor does not work correctly, the default light scene for all rooms is that both ceiling light groups are on.

Also the part of `NF1` which complements `NF2` is reflected by the assumption that the value of *outdoor_sensor*(room) remains constant if the sensor does not work correctly (OutdoorSensorReq). This assumption reflects that `NF1` is not a requirement on the controller, but on the way the sensor values are transmitted as input to the controller.
Anticipating the rule *Use_daylight* below we point out already here that the definition of *Activate_light_scene* contains a decision about the interpretation of requirement `U10`. Since nothing is said about what it means to maintain the ceiling light group on a given light scene, we interpret this as requesting that the ceiling lights are set to `minDimValue` if they do not enter explicitly the lights to be turned on for the given light scene (`U10`Req).

The control panel also allows to set the value T1 for a room (`U7`). This is formalizable by updates of a corresponding function t_1 – similarly for `FM4`, `FM5`. We group them into the *Set_parameters* part of the *Failure_and_service* submachine.

3.3 Automatic light control

The automatic control system is required to be able to switch on and off any light in a room or in a hallway section. Switching on is used to ensure that there is safe illumination in the room or hallway (`U1`, `U13`, `U14`). The lights in the hallway sections are not dimmable so that switching on can be done there only completely. Switching on is triggered by the two events (i) motion in the hallway (*somebody_is_moving*) and (ii) a door is open (*some_door_is_open*).

$$
\begin{array}{l}
Auto\_switch\_on\_in\_hallway(\mathrm{hallway}) = \\
\quad \textbf{if } (somebody\_is\_moving(\mathrm{hallway}) \lor some\_door\_is\_open(\mathrm{hallway})) \land \\
\qquad light\_is\_off(\mathrm{hallway}) \\
\quad \textbf{then} \\
\qquad Switch\_lights\_on(\mathrm{hallway})
\end{array}
$$

Switching on the light in a room is more complicated. According to `U3` and `U4` one has to distinguish two cases:

`U3` If the room is reoccupied within T1 minutes after the last person has left the room, the *chosen light scene* has to be reestablished.

`U4` If the room is reoccupied after more than T1 minutes since the last person has left the room, the *default light scene* has to be established.

In the first case, instead of establishing the *chosen light scene* we use the *last light scene* (`U3`Req) since otherwise the requirements would be incoherent, as the following example shows. By the definition in the dictionary in Part 4 of [Problem Description 2000], the *chosen light scene* is the scene selected with the control panel. Imagine the following scenario:

1. Person A enters the room and selects scene s.
2. Person A leaves the room for more than T1 minutes and no other person enters the room.
3. Person B enters the room. According to `U4`, the *default light scene* should be established.
4. Person B does not change the light scene and leaves the room.
5. Person B enters within T1 minutes. According to `U3`, we should establish the *chosen light scene* s.

As one can see Person B, upon entering the room for the first time, gets the *default light scene* and, upon reentering, gets the *chosen light scene* of Person A. This seems to be a flaw in [Problem Description 2000] and we therefore select the *last light scene*. For the reasons explained above we abstract from the value T1 and use the function *recently_occupied*.

$$
\begin{array}{l}
\mathit{Auto\_switch\_on\_in\_room}(\mathrm{room}) = \\
\quad \textbf{if } (\mathit{somebody\_is\_moving}(\mathrm{room}) \vee \mathit{some\_door\_is\_open}(\mathrm{room})) \wedge \\
\quad\quad \mathit{light\_is\_off}(\mathrm{room}) \\
\quad \textbf{then} \\
\quad\quad \mathit{Activate\_light\_scene}(\mathrm{room}, \mathrm{scene}) \\
\quad \textbf{where } \mathrm{scene} = \textbf{if } \mathit{recently\_occupied}(\mathrm{room}) \textbf{ then} \\
\quad\quad\quad\quad\quad\quad \mathit{last\_light\_scene}(\mathrm{room}) \\
\quad\quad\quad\quad\quad \textbf{else } \mathit{default\_light\_scene}(\mathrm{room})
\end{array}
$$

$$
\mathit{recently\_occupied}(\mathrm{room}) = \mathit{current\_time} - \mathit{last\_motion}(\mathrm{room}) \leq t_1(\mathrm{room})
$$

We do not commit here to any particular definition of *default_light_scene* (DefaultLightSceneReq). The definition in the dictionary is probably not reasonable because with that definition, requirement `U4` makes no sense.

The control system switches off the light in a room or in a hallway section if the location is not occupied for T3 or T2 minutes respectively (`FM2`, `FM3`). In accordance with requirement `NF5` we do not switch off the light in a hallway section if one of its buttons is defective. To reflect the malfunction condition `NF4`, we stipulate that occurrence of a malfunction for a motion sensor is interpreted as presence of motion so that the location appears as occupied.

$Auto\_switch\_off(\text{location}) =$
  if $location\_is\_hallway(\text{location}) \wedge any\_hallway\_button\_defect(\text{location})$
    then $skip$
    else
      if $\neg occupied(\text{location}) \wedge no\_motion\_for\_long\_time(\text{location}) \wedge$
        $\neg some\_door\_is\_open(\text{location}) \wedge \neg light\_is\_off(\text{location})$
      then $Switch\_lights\_off(\text{location})$

$no\_motion\_for\_long\_time(\text{location}) =$
  if $location\_is\_room(\text{location})$ **then**
    $current\_time - last\_motion(\text{location}) > t_3(\text{location})$
  else
    $current\_time - last\_motion(\text{location}) > t_2(\text{location})$

The control system should use daylight to achieve the desired *ambient light level* (`FM1`). We model this by reactivating the current light scene if the room is in ambient mode and there is no request for the ceiling lights. The following rule also reflects the informal need `U10`.

$Use\_daylight(\text{room}) =$
  if $no\_event\_for\_ceiling\_light(\text{room}) \wedge mode(\text{room}) = \texttt{Ambient}$ **then**
    $Activate\_light\_scene(\text{room}, last\_light\_scene(\text{room}))$

`U2` is fulfilled automatically in our requirements model because an ASM state remains unchanged unless a specific (user or control system) action triggers a change for the value of some specified functions for some specified arguments.

3.4 Failure and service

The last part of [Problem Description 2000] is about malfunctions. There are two actions to describe, namely identifying and handling malfunctions. Identifying malfunctions is a rather difficult application domain and not so much a software design problem. [Problem Description 2000] does not provide any further details on this issue so that we assume having a function *malfunction_occurs* telling whether a component works correctly or not; a component may be a hallway button, a light sensor, a motion sensor or any light. For building a concrete plant with its control software, this function has to be further specified by the customer, together with the support requested in `FM8` for finding the reasons for occuring malfunctions. To reflect the malfunction requirement `FM1` we stipulate that this function can also be updated manually.

```
Malfunction =
  forall component ∈ all_components
    if malfunction_occurs(component) then
      Handle_malfunction(component)
```

According to U8, FM7, and FM10, the handling of malfunction logs the corresponding information. In the case in which a hallway button is defective we switch on the lights in that hallway (NF5). In case a hallway motion detector is defective, by assumption (NF5bReq) the function *somebody_is_moving* is true and we therefore switch on the lights by rule *Auto_switch_on_in_hallway*. In the following rule we use i as index which has to match the name of the corresponding device (sensor, button)

```
Handle_malfunction(component) =
  case component of
    OutdoorLightSensor(i) →
      forall room ∈ rooms_under_lightsensor(i)
        Inform_user(room, LightSensorDefect(i))
        Inform_facility_manager(LightSensorDefect(i))
        Write_log_in_database(LightSensorDefect(i))
    MotionSensor(location, i) →
      if location_is_room(location) then
        Inform_user(location, MotionSensorDefect(location, i))
      Inform_facility_manager(MotionSensorDefect(location, i))
      Write_log_in_database(MotionSensorDefect(location, i))
    HallwayButton(hallway, i) →
      Switch_lights_on(hallway)
      Inform_facility_manager(HallwayButtonDefect(hallway, i))
      Write_log_in_database(HallwayButtonDefect(hallway, i))
    Luminaire(location, light) →
      Inform_facility_manager(LightDefect(location, light))
      Write_log_in_database(LightDefect(location, light))
```

To satisfy requirement NF9 we add a rule

Detect_unreasonable_input

Since [Problem Description 2000] contains no information on the meaning of NF9, we leave it to further refinement steps to provide a detailed definition, resulting from the discussion with the application domain expert who is supposed to know what inputs have to be considered as "unreasonable".

The system should provide reports on energy consumption (FM9). We formalize this requirement by introducing a dynamic function *dim_value* storing the current dim value of a light. Switching the light is defined as setting a dim value. If the dim value is less than 10% of the maximum dim value, then the light is switched off (see [Problem Description 2000], Table 2):

$$\begin{array}{l} \mathit{Switch\_light}(\mathrm{location}, \mathrm{light}, \mathrm{value}) = \\ \quad \textbf{if } \mathrm{value} < \mathtt{maxDimValue}/10 \textbf{ then} \\ \qquad \mathit{status\_of\_light}(\mathrm{location}, \mathrm{light}) := \mathtt{Light\_Off} \\ \quad \textbf{else} \\ \qquad \mathit{status\_of\_light}(\mathrm{location}, \mathrm{light}) := \mathtt{Light\_On} \\ \quad \mathit{dim\_value}(\mathrm{location}, \mathrm{light}) := \mathrm{value} \end{array}$$

Based on the values in $\mathit{dim\_value}$, we can define the function $\mathit{power\_consumption}$ computing the current power consumption. The function has to take into account the malfunctioning of lights:

$$\begin{array}{l} \mathit{power\_consumption} = \sum [p(\mathrm{l}, \mathit{dim\_value}(\mathrm{l})) \mid \mathrm{l} \in \mathit{dom}(\mathit{dim\_value})] \\ \quad \textbf{where } p(\mathrm{l}, \mathrm{v}) = \textbf{case } \mathit{light\_defect}(\mathrm{l}) \textbf{ of} \\ \qquad\qquad\qquad\qquad \mathtt{NotDefect} \rightarrow c * \mathrm{v} \\ \qquad\qquad\qquad\qquad \mathtt{DefectOn} \;\; \rightarrow c * \mathtt{maxDimValue} \\ \qquad\qquad\qquad\qquad \mathtt{DefectOff} \rightarrow c * \mathtt{minDimValue} \end{array}$$

We use the constant c to adjust the dim value to the electrical power. The energy consumption is the integral of the power consumption over the time. Therefore we store the power consumption in each step in a dynamic function and define the energy consumption as the product of the interval t_e with the sum of the power consumptions. We assume that the following rule will be executed every t_e minutes.

$$\begin{array}{l} \mathit{Report\_energy\_consumption} = \\ \quad \mathit{consumption}(\mathit{current\_time}) \qquad\qquad := \mathit{power\_consumption} \\ \quad \mathit{energy\_consumption}(\mathit{current\_time}) := t_e * \sum_t \mathit{consumption}(t) \end{array}$$

3.5 The requirements ground model

The model which results from the formalization in the preceding subsections is the following ASM, consisting of three submachines:

$$\begin{array}{l} \mathit{Light} = \mathit{Manual\_light\_control} \\ \qquad\quad\;\; \mathit{Automatic\_light\_control} \\ \qquad\quad\;\; \mathit{Failure\_and\_service} \end{array}$$

The $\mathit{Manual\_light\_control}$ actions were described in [Section 3.2]. We introduced rules for switching light on and off in hallways and in rooms and described the functionality of the control panel.

Manual_light_control =
  forall location $\in$ *all_locations*
    Manually_switch_off(location)
    if *location_is_room*(location) **then**
      Room_wall_button(location, `LightGroupWall`)
      Room_wall_button(location, `LightGroupWindow`)
      Control_panel(location, switch(location))
    if $\neg$*location_is_room*(location) **then**
      Hallway_button(location)

The second machine *Automatic_light_control* consists of automatically switching light on and off and using daylight to achieve the desired light level. Due to automatically switching lights on and off, we have also to observe the motion detectors. The rules are described in [Section 3.3].

Automatic_light_control =
  forall location $\in$ *all_locations*
    Auto_switch_off(location)
    Observe_motion_detector(location)
    if *location_is_room*(location) **then**
      Auto_switch_on_in_room(location)
      Use_daylight(location)
    if $\neg$*location_is_room*(location) **then**
      Auto_switch_on_in_hallway(location)

The last of the three submachines is *Failure_and_service*, containing handling of malfunctions, detecting unreasonable inputs, reporting energy consumption, and setting parameters (reflecting `FM4`, `FM5`, `FM11`, `U7`).

Failure_and_service =
  Malfunction
  Detect_unreasonable_input
  Report_energy_consumption
  Set_parameters

One can use different policies for the synchronization of the three machines of *Light* which has to guarantee the consistency of the three machines' update actions in the shared data area. One possibility is to make specific priority or scheduling assumptions on possibly conflicting actions, as we have indicated at various places during the formalization of the requirements. Another possibility is to impose a concrete scheduling on the coordination of the three submachines. Such a global policy relegates the consistency problem to the local levels of the single submachines. For our ground model, we can assume, for example for its executable version explained in the next section, that the manual and the automatic submachines alternate at a fixed rate – fast enough to guarantee the desired reaction time of the light system to user or environment input – and that the failure and service submachine is executed in between with a certain predescribed frequency, again determined by the time requirements for failure handling and general services. [Problem Description 2000] leaves all these issues completely open. In the ground model we could have reflected this freedom explicitly by introducing appropriate choice functions which determine at which time which submachine is running. For the executable version of our ground model we had to make some concrete realistic decisions.

We assume starting at an initial state in which all rooms and all hallways are empty and all lights are off. Especially, we assume the following initial values for our dynamic functions:

$$\begin{aligned} &mode(\text{room}) &&= \texttt{Manual} \\ &last\_light\_scene(\text{room}) &&= default\_light\_scene(\text{room}) \\ &last\_motion(\text{location}) &&= 0 \end{aligned}$$

4 Ground model validation

In this section we provide further details for the macros used in the previous sections which allow us to turn the ground model into an executable model which has been implemented in AsmGofer and is available electronically (see [Schmid 1999b]). This executable ground model version allows the customer to validate [Problem Description 2000] by experiments with our model. A preliminary version of this simulation model was presented in a demo at the Dagstuhl seminar on Requirements Capture [Börger et al. 1999b]. The definitions presented in this section have to be added to the requirements which occur in [Problem Description 2000] and can be viewed as further decisions made for directing the real (not any more prototypical) design. We skip the standard data structures needed to encode locations, lights, light groups, actuators, light sensor, etc.

$$\begin{aligned} &Inform\_user(\text{room}, \text{malfunction}) = \\ &\quad user\_information(\text{room}, current\_time, \text{malfunction}) := \texttt{True} \end{aligned}$$

$$\begin{aligned} &Inform\_facility\_manager(\text{malfunction}) = \\ &\quad facility\_manager(\text{malfunction}, current\_time) := \texttt{True} \end{aligned}$$

$$\begin{aligned} &Write\_log\_in\_database(\text{malfunction}) = \\ &\quad database(current\_time, \text{malfunction}) := \texttt{True} \end{aligned}$$

Typically the physical realization of *Inform_user* will be required to appear on the control panel display, but other solutions are possible.

There are several derived functions which can easily be defined and are needed for an executable version. Some of these definitions are listed below:

$$\begin{aligned}&\mathit{lightgroup\_is\_completely\_on}(\mathrm{room}, \mathrm{lg}) = \\ &\quad \forall\, \mathrm{l} \in \mathit{lights\_in\_group}(\mathrm{room}, \mathrm{lg})\,.\; \mathit{dim\_value}(\mathrm{room}, \mathrm{l}) = \mathtt{maxDimValue}\end{aligned}$$

$$\begin{aligned}&\mathit{light\_is\_on}(\mathrm{hallway}) = \\ &\quad \forall\, \mathrm{l} \in \mathit{lights\_at}(\mathrm{hallway})\,.\; \mathit{status\_of\_light}(\mathrm{hallway}, \mathrm{l}) = \mathtt{Light\_On}\end{aligned}$$

$$\begin{aligned}&\mathit{light\_is\_off}(\mathrm{location}) = \\ &\quad \forall\, \mathrm{l} \in \mathit{lights\_at}(\mathrm{location})\,.\; \mathit{status\_of\_light}(\mathrm{location}, \mathrm{l}) = \mathtt{Light\_Off}\end{aligned}$$

5 Conclusion

By capturing the Light Control Problem requirements as an ASM and making the ASM executable to support high-level simulation and debugging, we have shown how a piece of theory – the concept of ASM – can be used with practical advantage in a sensitive part of the software development process. The reader may wish to check that in elicitating, specifying and implementing the informal requirements, we have reflected all the requirements which are listed in the Problem Description, including the real-time aspects, except `U11`, `U12`, and `NF6-8` which are about norms and installation issues. We have disambiguated the requirements, removing inconsistencies and incoherencies, and have completed them through additional conditions which for documentation purposes are listed in the appendix. The semantic relevance of these additional requirements is different from that of those additional definitions which we have provided as refinements of the ground model to make it executable.

In this case study, structuring elements appear only in the form of parameterization of definitions and rules which exhibit the uniformity of certain requirements and make the specification reusable. In industrial applications, the situation is rather different; architectural requirements usually occupy a large place there [Hofmeister et al. 1999].

The AsmGofer code which implements our specification can be compiled to C++. The entire modeling and implementation effort, including most of the work to write up this paper, was half a person month.

Acknowledgement

We thank Dan Berry and three anonymous referees for critical comments on an earlier version of this paper.

A Additional requirements

`U1Req`	It is safe to allow a person who wants to rest in a room to choose a light scene in which all the lights are switched off and the room is dark.
`U3Req`	Instead of establishing the *chosen light scene* we use the *last light scene*.
`U10Req`	If the ceiling lights do not enter explicitly the lights to be turned on for the given light scene, they are set to `minDimValue`.
`NF5aReq`	Ceiling lights in a hallway section are "not controllable manually" if at least one hallway button is defective.
`NF5bReq`	If a motion detector is defective, its sensor value behaves as if there is motion.
PushButtonReq	Consistency of simultaneous pushing on different wall buttons (fixed priority or hardware solution).
RoomOccupationReq	A reasonable definition for a location to be not occupied is that there has been no motion for a period of max_quiet_time.
MotionDetectorReq	The motion sensor detects motion when users push buttons.
LightSceneReq	The function $lights\_to\_turn\_on$ computes an ordered set containing all lights that should be switched on together with their dim values. The order of the set is the order in which the lights should be turned on.
HallwayReq	The requirements `FM1` and `NF3` are useless for hallways if these are without windows.
OutdoorSensorReq	The sensor value of an outdoor light sensor remains constant if the sensor does not work correctly.
DefaultLightSceneReq	We do not commit to any particular definition of $default\_light\_scene$.

References

[Berry 1995] Berry, D. M. (1995), "The importance of ignorance in requirements engineering", *Journal of Systems and Software*, 28(2):179–184.

[Börger 1999] Börger, E. (1999), "High level system design and analysis using Abstract State Machines", In Hutter, D., Stephan, W., Traverso, P., and Ullmann, M., editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in Lecture Notes in Computer Science, pages 1–43, Springer-Verlag.

[Börger et al. 1999a] Börger, E., Hörger, B., Parnas, D., and Rombach, D. (1999), "Requirements Capture, Documentation and Validation", Web pages at: http://www.iese.fhg.de/Dagstuhl/seminar99241.html.

[Börger et al. 1999b] Börger, E., Riccobene, E., and Schmid, J. (1999), "Software requirements specification of the Light Control System", In Börger, E., Hörger, B., Parnas, D., and Rombach, D., editors, *Requirements Capture, Documentation, and Validation*, Dagstuhl Seminar No. 99241.

[Börger and Schmid 2000] Börger, E. and Schmid, J. (2000), "Composition and submachine concepts", In *Computer Science Logic (CSL 2000)*, Lecture Notes in Computer Science, to appear.

[Customer Feedback 1999] "The Light Control Case Study: Customer Feedback", Web pages at: http://rn.informatik.uni-kl.de/~recs/qna.

[Gurevich 1995] Gurevich, Y. (1995), "Evolving Algebras 1993: Lipari Guide", In Börger, E., editor, *Specification and Validation Methods*, pages 9–36, Oxford University Press.

[Hofmeister et al. 1999] Hofmeister, C., Nord, R., and Soni, D. (1999), "Applied software architecture".

[Parnas 1999] Parnas, D. (1999), "Formal methods technology transfer will fail", *Journal of Systems and Software*, 40(3):195–198.

[Parnas and Madey 1995] Parnas, D. and Madey, J. (1995), "Functional documents for computer systems", *Science of Computer Programming*, 25(1):41–61.

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

[Schmid 1999a] Schmid, J. (1999), "Executing ASM specifications with AsmGofer", Web pages at: http://www.tydo.de/AsmGofer.

[Schmid 1999b] Schmid, J. (1999), "The light control system", Web page at: http://www.tydo.de/AsmGofer/light.