A Method for Systematic Requirements Elicitation: Application to a Light Control System

Jeanine Souquières[1] and Maritta Heisel[2]

[1] LORIA—Université Nancy2, B.P. 239 Bâtiment LORIA, F-54506Vandœuvre-les-Nancy
[2] Fakultät für Informatik, Universität Magdeburg, D-39016 Magdeburg

Abstract. This paper demonstrates the use of a systematic approach to clarify and analyze the requirements of a light control system. The method includes a formalization of the requirements and the analysis of interactions between them.

1 Introduction

Because it is crucial for the success of software projects that the requirements be correct, unambiguous and complete, a systematic approach to requirements elicitation that helps to detect inconsistencies, incompletnesses and ambiguities is of great value. We have developed a method to perform the early phases of the software lifecycle in a systematic way [HS99a,HS99b]. The method supports two phases, namely requirements elicitation and specification development. In this paper, we demonstrate the usefulness of our method by way of applying it to the case study of a light control system.

The requirements elicitation process begins with a reflection on the application domain and an informal description of the customer needs. It consists in taking into account elements of an initial informal requirements document and analyzing them exhaustively in order to obtain a coherent and complete set of requirements. In particular, the objectives of our method for requirements elicitation are to

- understand the problem,
- fix the used vocabulary,
- disambiguate the requirements,
- find incoherences and missing requirements,
- establish an adequate starting point for a formal specification of the software system.

We use formal techniques very early in the development process in order to analyze the customer needs in a detailed way and to reveal inconsistencies, incompletnesses and ambiguities in the requirements. Formal techniques are very helpful here, because they enforce a level of rigor and detail that is not achieved using only informal techniques. In particular, an analysis of possible interactions between requirements is performed all along this formalization process. An interaction is a situation where the requirements make perfect sense when considered in isolation, but their combination leads to contradictions or unwanted or unexpected system behavior.

Requirements should be stated as *fragments*. Such fragments can correspond to parts of independent scenarios of system behavior.

The starting point of the requirements elicitation phase is a requirements document provided by the customer. Its result includes an updated requirements document, a summary of the domain vocabulary, and several formal documents.

Section 2 gives an overview of the requirements elicitation method. Section 3 discusses related work. Section 4 illustrates the application of our method to the light control case study. Section 5 presents a discussion of the method and its benefits.

2 Method for Requirements Elicitation

Our approach is inspired by the work of Jackson and Zave [JZ95,ZJ97] and by the first steps of object oriented methods and notations such as Fusion [CAB$^+$94] or UML [BRJ99]. It starts with a brainstorming process where the application domain and the requirements are described in natural language. This informal description is then transformed into a formal representation. On the formal representation, interaction analyses are performed. Their purpose is to obtain a consistent and adequate set of requirements. The method for requirements elicitation is described in [HS99a,HS99b], the algorithm to detect interactions is described in [HS98]. In the following, we give a brief overview of the main steps of the method.

Step 1: Introduce the domain vocabulary. The different notions of the application domain are expressed in a textual form.

Step 2: State the facts, assumptions, and requirements concerning the system as a set of fragments corresponding to parts of scenarios of system behavior. It does not suffice to just state requirements for the system. Often, facts and assumptions must be introduced to make the requirements satisfiable. *Facts* express conditions that always hold in the application domain, regardless of the implementation of the software system. Other requirements cannot be enforced, because, e.g., human users might violate regulations. Such regulations are expressed as *assumptions*.

Step 3: List all relevant events that can happen in connection with the system, and classify them. Events concern the reactive part of the system. The classification we adopt is the one proposed by Jackson et Zave [JZ95]. It must be stated who is in control of the event (the software system or its environment) and who can observe it.

Step 4: List the system operations that can be invoked by users. This step concerns the non-reactive part of the system to be described. For purely reactive systems, it can be empty. System operations are usually independent of the physical components of the system, but refer to the information that is stored in the part of the system to be realized by software.

Steps 1 through 4 can be carried out in any order or in parallel, with repetitions and revisions. There are *validation conditions* associated with the different steps, supporting quality assurance of the resulting product. They state necessary semantic conditions that the developed artifact must fulfill in order to serve its purpose properly. For Steps 1–4, the validation conditions are:

– The vocabulary must contain exactly the notions occurring in the facts, assumptions, requirements, operations, and events.
– There must not be any events controlled by the software system and not shared with the environment.

Step 5: Formalize the facts, assumptions, and requirements as constraints on the possible traces of system events. In this step, the facts, assumptions, and requirements must be formalized one by one. But before the formalized constraint is added to the set of already accepted constraints, its possible interactions with them should be analyzed, in order to detect inconsistencies or undesired behavior [HS98]. Step 5 consists of six sub-steps.

Step 5.1: Formalize the new constraint as a predicate on system event traces. To formalize facts, assumptions and requirements, we use traces, i.e., sequences of events happening in a given state of the system at a given time. The system is started in state S_1. When event e_1 happens at time t_1, then the system enters the state S_2, and so forth:

$$S_1 \xrightarrow[t_1]{e_1} S_2 \xrightarrow[t_2]{e_2} \ldots\ S_n \xrightarrow[t_n]{e_n} S_{n+1}\ \ldots$$

Let be Tr the set of possible traces. A constraint is expressed as a formula restricting the set Tr. For a given trace $tr \in Tr$, $tr(i)$ denotes the i-th element of this trace, $tr(i).s$ the state of the i-th element, $tr(i).e$ the event which occurs in that state, and $tr(i).t$ is the time at which the event occurs. For each possible trace, its prefixes are also possible traces. A formal specification of traces is given in Appendix A.

Sometimes, it may be necessary to introduce *predicates* on the system state to be able to express the constraints formally. For each predicate, events that establish it and events that falsify it must be given. These events must be shared with the software system.

In the following, we use the term *literal* to mean predicate or event symbols, or negations of such symbols. An event symbol e is supposed to mean "event e occurs", whereas $\neg\ e$ is supposed to mean "event e does not occur". If we refer to predicate symbols and their negations, we will use the term *predicate literal*. *Event literals* are defined analogously.

We recommend to express – if possible – constraints as implications, where either the precondition of the implication refers to an earlier state or an earlier point in time than the postcondition, or both the pre- and postcondition refer to the same state, i.e. we state an invariant of the system.

Example. If a person occupies a room, there has to be safe illumination.

$$\begin{array}{l} \forall\, tr : Tr;\ room : ROOM \bullet \forall\, i : \mathsf{dom}\ tr \bullet \\ \quad occupied(tr(i).s, room) \Rightarrow safe\_illumination(tr(i).s, room) \end{array}$$

The expression $\mathsf{dom}\ tr$ denotes the valid indices of the trace, i.e., $1 \mathrel{..} \#tr$, where $\#tr$ denotes the length of the trace. The symbols *occupied* and *safe_illumination* are predicate symbols parameterized with the system state and a room.

Step 5.2: Give a schematic expression of the constraint. We have defined an algorithm that determines interaction candidates for a new constraint with respect to a set of already accepted constraints, which is described in Step 5.4. This algorithm uses schematic expressions of formalized constraints. These schematic expressions have the form

$$x_1 \diamond x_2 \diamond \ldots \diamond x_n \rightsquigarrow y_1 \diamond y_2 \diamond \ldots \diamond y_k$$

where the x_i, y_j are literals, and each $\diamond$ denotes conjunction or disjunction. The symbol "$\rightsquigarrow$" separates the precondition from the postcondition.

For transforming a constraint into its schematic form, we abstract from quantifiers and from parameters of predicate and event symbols.
Example. The schematic expression corresponding to the above constraint is

$$occupied \rightsquigarrow safe\_illumination$$

Step 5.3: Update tables of semantic relations. The detection of constraint interactions cannot be based on syntax alone. We also must take into account the semantic relations between the different symbols. A predicate may imply another predicate, an event may only be possible if the system state fulfills a predicate, and for each predicate, we must know which events establish and which events falsify it. We construct three tables of semantic relations:

1. Necessary conditions for events. If an event e can only occur if predicate literal pl is true, then this table has an entry $pl \leftsquigarrow e$.
   Example. The event $leave\_last$ can only occur if the room is occupied: $occupied \leftsquigarrow leave\_last$
2. Events establishing predicate literals. For each predicate literal pl, we need to know the events e that establish it: $e \rightsquigarrow pl$
   Example. The predicate $occupied$ is established by the event $enter\_first$: $enter\_first \rightsquigarrow occupied$
3. Relations between predicate literals. For each predicate symbol p, we determine
   - the set of predicate literals it entails: $p_{\Rightarrow} = \{q : PLit \mid p \Rightarrow q\}$.
   - the set of predicate literals its negation entails: $\neg\, p_{\Rightarrow} = \{q : PLit \mid \neg\, p \Rightarrow q\}$

   Example. $occupied_{\Rightarrow} = \{safe\_illumination\}$

These tables are not only useful to detect interactions; they are also useful to develop and validate a formal specification of the software system.

Step 5.4: Determine interaction candidates. Interaction candidates are calculated from the list of schematic requirements set up in Step 5.2 and the tables of semantic relations constructed in Step 5.3. Two constraints are interaction candidates for one another if they have common preconditions but incompatible postconditions, as is illustrated in Figure 1. "Incompatible" does not necessarily mean "logically inconsistent"; it could also mean "inadequate" for the purpose of the system.

Our algorithm to determine interaction candidates consists of two parts: precondition interaction analysis determines constraints with preconditions that are neither

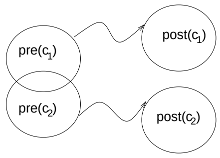

Fig. 1. Interaction candidates

exclusive nor independent of each other. This means, there are situations where both constraints might apply. The postconditions of such constraints have to be checked for incompatibility. Postcondition interaction analysis, on the other hand, determines as candidates those constraints with incompatible postconditions. If in such a case the preconditions do not exclude each other, an interaction occurs.

Precondition Interaction Analysis. If two constraints[1] $\underline{x} \rightsquigarrow \underline{y}$ and $\underline{u} \rightsquigarrow \underline{w}$ have common literals in their precondition ($\underline{x} \cap \underline{u} \neq \varnothing$), then they are certainly interaction candidates. But the common precondition may also be hidden. For example, if $\underline{x}$ contains the event e, $\underline{u}$ contains the predicate literal pl, and e is only possible if pl holds ($pl \leftsquigarrow e$), then we also have detected a common precondition pl of the two constraints.

The common precondition may also be detected via reasoning on predicates. If, for example, $\underline{x}$ contains the predicate literal pl, $\underline{u}$ contains the predicate literal q, and there is a predicate literal w with $pl \Rightarrow w$ and $q \Rightarrow w$, then w is a common precondition.

Figure 2 illustrates the definition of the set $C_{pre}(c', far)$ of precondition interaction candidates of a new constraint c' with respect to the set far of facts, assumptions, and requirements already defined.

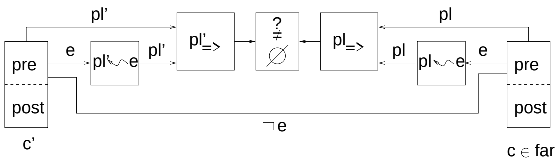

Fig. 2. Determining interaction candidates by precondition analysis

Let

$$precond(x_1 \diamond x_2 \diamond \ldots \diamond x_n \rightsquigarrow y_1 \diamond y_2 \diamond \ldots \diamond y_k) = \{x_1, \ldots x_n\}$$

The set $pre\_predicates(c)$ of predicates that hold in the precondition of a constraint c are the predicate literals $pl \in precond(c)$ and the predicate literals pl with $pl \leftsquigarrow e$, for all event symbols $e \in precond(c)$:

$$\leftsquigarrow e = \{pl : PLit \mid pl \leftsquigarrow e\}$$
$$pre\_predicates(c) = (precond(c) \cap PLit) \cup \bigcup_{e \in precond(c) \cap EVENT} \leftsquigarrow e$$

[1] Underlined identifiers denote sets of literals.

The *predicative closure* of the precondition of a constraint c results from the transitive and reflexive closure of the set $pre\_predicates(c)$ with respect to implication:

$$\bigcup_{pl \in pre\_predicates(c)} pl_{\Rightarrow}$$

A constraint $c \in far$ is an interaction candidate for a new constraint c' if their preconditions or their respective predicative closures contain common literals.

$$\begin{aligned}
&C_{pre}(c', far) = \\
&\quad \{c : far \mid precond(c) \cap precond(c') \neq \varnothing\} \\
&\quad \cup \\
&\quad \{c : far \mid \exists\, pl : pre_predicates(c);\ pl' : pre_predicates(c') \bullet pl_{\Rightarrow} \cap pl'_{\Rightarrow} \neq \varnothing\}
\end{aligned}$$

Two cases must be distinguished, because the precondition of a constraint can contain event literals, whereas the predicative closure of the precondition only contains predicate literals.

From the definition of $C_{pre}(c', far)$, it follows that the set of candidates is independent of the order in which the constraints are added, provided that the same tables of semantic relations are used to compute $\leftsquigarrow e$ and $pl_{\Rightarrow}$. Moreover, the candidate function distributes over set union of the preconditions of constraints.

Hence, when a constraint is changed by adding a new literal to its precondition, a new interaction analysis has to be performed only on this new literal.

Postcondition Interaction Analysis. To find conflicting postconditions, we compute the predicative closure of the postcondition of the new constraint c' and the one of each constraint $c \in far$ in much the same way as for the preconditions.

$$\begin{aligned}
&postcond(x_1 \diamond x_2 \diamond \ldots \diamond x_n \rightsquigarrow y_1 \diamond y_2 \diamond \ldots \diamond y_k) = \{y_1, \ldots y_k\} \\
&e_{\rightsquigarrow} = \{pl : PLit \mid e \rightsquigarrow pl\} \\
&post_predicates(c) = (postcond(c) \cap PLit) \cup \bigcup_{e \in postcond(c) \cap EVENT} e_{\rightsquigarrow}
\end{aligned}$$

A constraint c is an interaction candidate for the new constraint c' if there exists a literal pl in its postcondition or in its predicative closure, the negation of which is in the postcondition of c' or in its predicative closure. Figure 3 illustrates the definition.

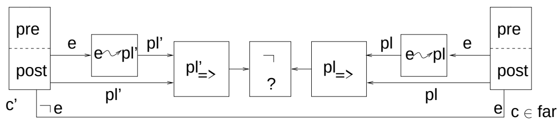

Fig. 3. Determining interaction candidates by postcondition analysis

$C_{post}(c', far) =$
$\{c : far \mid postcond(c)\ opposite\ postcond(c')\}$
$\cup$
$\{c : far \mid \exists\, pl : post\_predicates(c);\ pl' : post\_predicates(c') \bullet pl_{\Rightarrow}\ opposite\ pl'_{\Rightarrow}\}$
$ls_1\ opposite\ ls_2 \Leftrightarrow \exists\, x : ls_1 \bullet \neg\, x \in ls_2$

where ls_1, ls_2 are sets of literals, and $\neg\,\neg\, l = l$.

This definition is symmetric, too, and C_{post} distributes over set union of postconditions of constraints.

The set of interaction candidates $C(c', far)$ of a new constraint c' with respect to the set far is the union of the precondition and the postcondition interaction candidates:

$$C(c', far) = C_{pre}(c', far) \cup C_{post}(c', far)$$

Step 5.5: Decide if there are interactions of the new constraint with the candidates determined in Step 5.4. The algorithm determines a set of candidates to examine. It does not prove that an interaction exists between the new constraint and each candidate. It is up to the analyst and the customer to decide if the conjunction of the new constraint with the candidates yields an unwanted behavior or if it is even contradictory.

Step 5.6: Resolve interactions. To resolve an interaction, we usually relax requirements or strengthen assumptions. Once a constraint has been modified, an interaction analysis on those literals that were changed or newly introduced must be performed.

The following validation conditions are associated with Step 5:
- each requirement of Step 2 must be expressed,
- the set of constraints must be consistent,
- for each introduced predicate, the events that modify it must be observable by the software system.

Steps 5.1 to 5.6 preserve the mutual coherence between the different constraints. Usually, revisions and communication with customers will be necessary.

3 Related work

Before we use our method to analyze the requirements of a light control system, we contrast our work to other requirements engineering approaches.

Separating domain knowledge from requirements and checking consistency between these two is frequent in the literature. For example, when Parnas describes the mathematical content of a requirements document for a nuclear shutdown system [Par95], he introduces different mathematical relations, one describing the environment of the computer system, and one describing its requirements. The two relations must fulfill certain feasibility conditions. Parnas also notes that a critical step in documenting the requirements concerns the identification of the environmental quantities to be measured or controlled and the representation of those quantities by mathematical variables. He proposes to characterize environmental quantities as either monitored or controlled. This distinction corresponds to the classification of events in our approach and the one of Jackson and Zave [JZ95]. Reading monitored quantities corresponds to events observable by the software system, and controlling quantities corresponds to events controlled by the software system.

Whereas assumptions are used neither by Parnas nor by Jackson and Zave, they do play a role in KAOS [DL96]. KAOS supports the design of composite systems (see also [FFH91]). Such a system consists of human, software and hardware agents, each of them being assigned responsibility for some goals. The KAOS approach is goal-oriented: goals are stated and then elaborated into KAOS specifications in several consecutive steps. That process starts with elaborating the goals in an AND/OR structure and ends with assigning responsibilities to agents. KAOS is similar to our approach in that it provides heuristics for requirements elicitation and specification development. However, it uses its own language and also takes a much broader perspective: not only the software system, but also its environment are modeled in detail.

Easterbrook and Nuseibeh [EN96] do not distinguish different phases for requirements elicitation and specification development. They elicit more requirements when they detect inconsistencies in their specifications. On the one hand, their approach makes it possible to delay the resolution of conflicts. On the other hand, this kind of "lazy" requirements elicitation delays the point where a definite contract between customers and providers of the software product can be made. Moreover, it is harder to validate a specification with respect to the requirements when no separate requirements document is set up that is checked for consistency and completeness in its own right.

Although our method is inspired by Jackson and Zave's approach to requirements engineering [JZ95,ZJ97], there are some important differences. First, they consider a specification to be a special kind of requirement. Requirements do not make statements about the state of the software system. In contrast to this view, we consider a specification to be a *model* of the software system to be built in order to satisfy the requirements. It forms the basis for refinement and implementation. Therefore, a specification may – in contrast to the requirements – make statements about the software system that are not directly observable by the environment. Second, the most important part of our approach is the methodological guidance that is given to analysts and specifiers, as well as the validation of the developed products. Jackson and Zave do not address these issues explicitly.

Of the object-oriented methods, we have only borrowed the very first steps, where the relevant vocabulary is introduced in a systematic way. Our approach is not biased toward object-oriented development.

4 Application of the Method to a Light Control system

We now demonstrate the application of the method described in Section 2. The task is to develop a new light control system for a university building in Kaiserslautern. There are two different groups of persons who have different requirements, namely the *users* of offices and the *facility manager* who is in charge of the technical equipment of the building. A detailed *problem description* has been published under http://rn.informatik.uni-kl.de/~recs. This document is divided into four parts: a problem introduction, the architecture of the building, the informal needs, and a dictionary. In this paper, we concentrate on the requirements concerning offices, which are just called “rooms” in the following. The requirements for hallways, laboratories, and other kinds of rooms can be treated analogously. Although we use the numbering of requirements as given in the problem description, this section is self-contained and can be read without having read the problem description.

4.1 Informal Steps

Step 1: Introduce the domain vocabulary. Figure 4 is a graphical representation of the domain vocabulary we have used to carry out the case study. Boxes denote entities, and diamonds denote relations. Arrows indicate the direction of the relations. The relation *has* denotes static aspects of the system, whereas all other relations concern dynamic aspects, expressed by events.

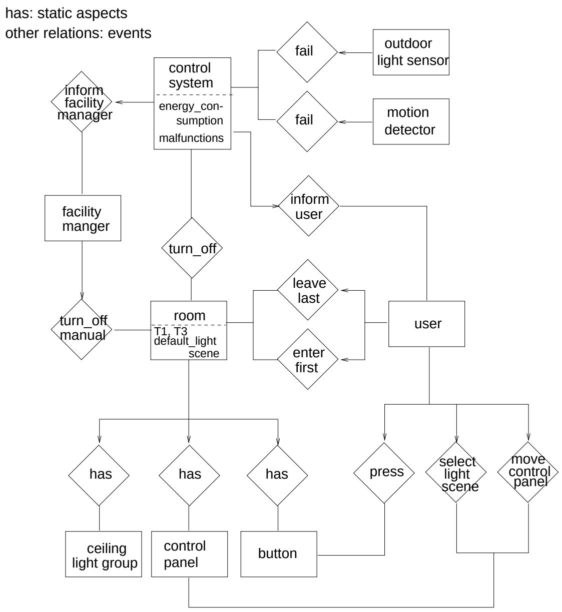

Fig. 4. Domain vocabulary for the light control system

Step 2: State the facts, assumptions, and requirements. We will consider the following requirements:

- *U1.* If a person occupies a room, there has to be safe illumination, if nothing else is desired by the chosen light scene.
- *U2.* As long as a room is occupied, the chosen light scene has to be maintained.
- *U3.* If the room is re-occupied within $T1$ minutes after the last person has left the room, the chosen light scene has to be reestablished.

- *U5.* For each room, the chosen light scene can be set by using the room control panel.
- *U6.* For each room, the default light scene can be set by using the room control panel.
- *U7.* For each room, the value $T1$ can be set by using the room control panel.
- *U8.* If any outdoor light sensor or the motion detector of a room does not work correctly, the user of this room has to be informed.
- *U9.* The room control panel should contain at least (i) a possibility to set each ceiling light group, (ii) a possibility to set the chosen and the default light scene, and (iii) a possibility to set $T1$.
- *U10.* The ceiling light groups should be maintained by the control system depending on the current light scene.
- *U11.* A room control panel should be movable as is a telephone.
- *FM1.* Use daylight to achieve the desired light setting of each room whenever possible.
- *FM3.* The ceiling light groups in a room have to be off when the room is unoccupied for at least $T3$ minutes.
- *FM5* For each room, the value $T3$ can be set by using the facility manager control panel.
- *FM6.* The facility manager can turn off the ceiling light groups in a room that is not occupied.
- *FM7.* If a malfunction occurs, the facility manager has to be informed.
- *FM8.* If a malfunction occurs, the control system supports the facility manager in finding the reason.
- *FM9.* The system provides reports on current and past energy consumption.
- *FM10.* All malfunctions and unusual conditions are stored and reported on request.
- *FM11.* Malfunctions that the system cannot detect can be entered manually.

Step 3: List all relevant events that can happen in connection with the system, and classify them.

- One event is controlled by the environment and not shared with the software system, namely $move\_control\_panel$.
- Events controlled by the environment and shared with the software system:
  - $enter\_first$: the first person enters the room, making the room occupied.
  - $leave\_last$: the last person leaves the room, making the room unoccupied.
  - $press$: a push button in a room is pressed.
  - $select\_light\_scene$: a light scene is selected using the room control panel.
  - $turn\_off\_manual$: the facility manager turns off the lights.
  - $fail$: a device fails.
- Events controlled by the software system and shared with the environment:
  - $turn\_off$: the control system turns off the lights in a room.
  - $inform\_user$, $inform\_facility\_manager$: the users and the facility manager are informed in case of a device failure.
- As required by the validation condition, there is no event controlled by the software system and not shared with the environment.

Operation	motivated by
$set\_default\_light\_scene$	U6
set_T1	U7
set_T3	FM5
$find\_reason\_for\_malfunctions$	FM8
$report\_energy\_consumption$	FM9
$report\_malfunctions$	FM10
$enter\_malfunctions$	FM11

Table 1. System operations for the light control system

Step 4: List the system operations that can be invoked by the users. They concern the transformational part of the system. An overview is given in Table 1.

Details of these functions are not specified in the requirements elicitation phase but in the specification phase, and are not given in this paper.

4.2 Formalization and Interaction Analysis

In the following, we carry out Step 5 of our method in some detail for a subset of the requirements concerning rooms. For each formalized requirement, we discuss unclear points and decisions taken.

Requirement U1. If a person occupies a room, there has to be safe illumination, if nothing else is desired by the chosen light scene.
Formalization:

$$\forall\, tr : Tr;\ room : ROOM \bullet \forall\, i : \mathsf{dom}\ tr \bullet \\ \quad occupied(tr(i).s, room) \Rightarrow safe\_illumination(tr(i).s, room)$$

Two predicates on the system state have been introduced, $occupied$ and $safe\_illumination$, both parameterized with a room. To enhance readability of the constraints, we only state the requirements for single rooms, i.e., we will not mention rooms as parameters any more in the following.

Schematic expression: $occupied \leadsto safe\_illumination$

Semantic relations: first, we specify which events modify the two new predicates.

$$\begin{array}{rl} enter\_first & \leadsto occupied \\ leave\_last & \leadsto \neg\ occupied \end{array} \qquad \begin{array}{rl} enter\_first & \leadsto safe\_illumination \\ turn\_off & \leadsto \neg\ safe\_illumination \\ turn\_off\_manual & \leadsto \neg\ safe\_illumination \end{array}$$

Second, the predicate $occupied$ and its negation are necessary conditions for some events:

$$\begin{array}{llll} \neg\, occupied & \leftharpoondown\!\!\sim enter\_first & occupied & \leftharpoondown\!\!\sim press \\ occupied & \leftharpoondown\!\!\sim leave\_last & occupied & \leftharpoondown\!\!\sim select\_light\_scene \end{array}$$

Third, we define: $occupied_{\Rightarrow} = \{safe\_illumination\}$

Discussion. Note that the formalization does not contain an equivalent of the phrase "if nothing else is desired by the chosen light scene". It is not clear what is meant by this phrase. Hence, we have detected an *ambiguous requirement*. In a real project, customers would be asked for clarification.

As it stands now, the requirement is quite strong: from the formalized constraint, it follows that the light must automatically be switched on when somebody enters the room, and that it is even impossible to switch off the light using a push button if there is not enough daylight for safe illumination.

In the following, our interaction detection algorithm will detect inconsistencies of this formalization with other requirements, which would probably cause us to weaken the constraint in a real project.

Requirement U2. As long as a room is occupied, the chosen light scene has to be maintained.
Formalization: $\forall\, tr : Tr \bullet occupied(tr(i).s) \Rightarrow tr(i).e \neq turn\_off$

Schematic expression: $occupied \rightsquigarrow \neg\, turn\_off$

Semantic relations: none, because no new symbols are introduced with the formalization of U2.

Interaction candidates: U1 is an interaction candidate, because U1 and U2 have the same precondition. There is no interaction, because the conjunction of postconditions is a desired behavior.

Discussion. Requirement U2 is ambiguous, too. It is not clear what is meant by the word "maintain". What if the user pushes a button? According to the dictionary (Part 4 of the problem description), light scenes are chosen using the room control panel. If pushing a button does not change the chosen light scene, what effect is desired of this event?

We have decided to interpret U2 meaning that the *control system* is not allowed to change the light scene established in the room. Since the only event that could be used by the system to do so is $turn\_off$, our formalization says that this event is not allowed to occur while the room is occupied.

Requirement U3. If the room is re-occupied within $T1$ minutes after the last person has left the room, the chosen light scene has to be reestablished.
Formalization: we introduce a function that denotes the light scene required for a room in a given state: $light\_scene : STATE \rightarrow LIGHT\_SCENE$

$$\forall\, tr : Tr \bullet \forall\, i, j : \mathrm{dom}\, tr \mid j > i \land j + 1 \in \mathrm{dom}\, tr \bullet$$
$$tr(i).e = leave\_last \land tr(j).e = enter\_first$$
$$\land (\forall\, k : i \mathinner{..} j \bullet \neg\, occupied(tr(k).s)) \land tr(j).t - tr(i).t \leq T1$$
$$\Rightarrow light\_scene(tr(j+1).s) = light\_scene(tr(i).s)$$

The premise of the formula captures the fact that the interval in which the room is not occupied is marked by the events $leave\_last$ and $enter\_first$. If these occur within $T1$ time units, the light scene afterwards must be the same as before.

Schematic expression:
$leave\_last \land enter\_first \land \neg\, occupied \land lesseqT1 \leadsto same\_light\_scene$

Two new predicates have been introduced, $lesseqT1$ and $same\_light\_scene$ to avoid functions in schematic expressions.

Semantic relations[2]: it is clear that we should have the following relations:

$$\begin{aligned} enter\_first &\leadsto same\_light\_scene \\ select\_light\_scene &\leadsto \neg\, same\_light\_scene \end{aligned}$$

However, it is not obvious if pressing a button or turning off the lights (either manually or automatically) changes the required light scene or not. Therefore, the following semantic relations must be discussed with the customers:

$$\begin{aligned} press &\leadsto \neg\, same\_light\_scene \\ turn\_off &\leadsto \neg\, same\_light\_scene \\ turn\_off\_manual &\leadsto \neg\, same\_light\_scene \end{aligned}$$

Interaction candidates: U1 and U2 are candidates, because $occupied \leftsquigarrow leave\_last$ (i.e., $occupied$ is a common precondition of the three constraints), but there is no interaction.

Discussion. The formalization of U3 shows how real-time requirements can be treated using our method. The requirement itself seems clear to us, but it is not clear how a light scene required for a room can be changed, see discussion of Requirement U2. Hence, the relations between the predicate $same\_light\_scene$ and the different events are unclear.

Requirement U5. For each room, the chosen light scene can be set by using the room control panel.
Formalization:

$$\forall\, tr : Tr;\ ls : LIGHT\_SCENE \bullet \forall\, i : \mathrm{dom}\, tr \mid i < \#tr \bullet$$
$$tr(i).e = select\_light\_scene(ls) \Rightarrow light\_scene(tr(i+1).s) = ls$$

Schematic expression: $select\_light\_scene \leadsto selected\_light\_scene\_established$

[2] We do not give events establishing or falsifying $lesseqT1$, because the passing of time needs no particular events.

Semantic relations:

$$\begin{aligned} \mathit{select\_light\_scene} &\leadsto \mathit{selected\_light\_scene\_established} \\ \mathit{turn\_off} &\leadsto \neg\, \mathit{selected\_light\_scene\_established} \\ \mathit{turn\_off\_manual} &\leadsto \neg\, \mathit{selected\_light\_scene\_established} \end{aligned}$$

Interaction candidates: because the event $\mathit{select\_light\_scene}$ is only possible if the predicate $\mathit{occupied}$ holds, U1–U3 are interaction candidates. We judge that there is no interaction with U2 and U3, but there might be an interaction with U1: what if the selected light scene does not guarantee safe illumination? Is this possible at all? These questions need clarification.

Requirements U6 and U7. For each room, the default light scene / the value $T1$ can be set by using the room control panel.

Although the wording is almost the same as for requirement U5, we treat U6 and U7 differently from U5. We interpret U5 in such a way that setting a current light scene is not fundamentally different from pushing buttons to switch on or off some light group. Hence, U5 concerns the reactive part of the system.

In contrast, the default light scene and the value $T1$ can be regarded as an attributes of a room. Therefore, U6 and U7 concern the transformational part of the system. In Step 4, we have introduced the system operations $\mathit{set\_default\_light\_scene}$ and $\mathit{set\_T1}$. The definition of these functions is not part of the requirements elicitation phase of our method, but of the specification phase, see [HS99a].

Requirement U8. If any outdoor light sensor or the motion detector of a room does not work correctly, the user of this room has to be informed.
Formalization:

$$\begin{aligned} &\forall\, d : \mathit{DEVICE} \bullet \mathit{outdoor\_light\_sensor}(d) \vee \mathit{motion\_detector}(d) \\ &\quad \Rightarrow \mathit{fail}(d) \;\; \mathit{immediately\_followed\_by} \;\; \mathit{inform\_user}(d) \end{aligned}$$

This formalization uses a type DEVICE with predicates $\mathit{outdoor\_light\_sensor}$ and $\mathit{motion\_detector}$ (the definition of $\mathit{immediately\_followed\_by}$ is given in Appendix A).

Schematic expression: $\mathit{fail} \leadsto \mathit{inform\_user}$

Semantic relations: none.

Interaction candidates: none.

Discussion. The requirement says nothing about how a sensor failure is detected (this information should be given in Section 2.8 of the problem description, where the sensors are described) and how the user is to be informed.

Requirements U9 and U11 make statements about the functionality of the control panel, e.g., that it should be movable and that it should contain the possibility to set the chosen light scene.

These requirements concern the hardware to be purchased. In our method, they are reflected in Step 3 when we list the possible events: only if U9 and U11 are fulfilled, we can introduce events like $move\_control\_panel$ and $select\_light\_\ scene$. However, we propose to organize the requirements document in a different way: hardware requirements should be stated in a separate section.

Requirement U10. The ceiling light groups should be maintained by the control system depending on the current light scene.
Formalization:
$\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet conforms(light\_state(tr(i).s), light\_scene(tr(i).s))$

Here, we have introduced a new function $light\_state$ that yields the state of the luminaries in the room. The predicate $conforms$ is intended to mean that the "light state" of the room is such that the ambient light level specified by the current light scene is achieved.

Schematic expression: $true \leadsto conforms$

Semantic relations: none.

Normally, we should give events that establish and falsify the predicate $conforms$. However, we require $conforms$ to be an invariant of the system. Hence, there are no events that falsify it. There also are no specific events that establish it, because the underlying idea of the system design is that the state of the luminaries should always coincide with the chosen light scene.

Interaction candidates: because $p \Rightarrow true$ for all predicates p, we get as interaction candidates all requirements formalized so far, i.e., U1–U3, U5, U8. As for U5, there might be an interaction with U1. How exactly can safe illumination be guaranteed?

Discussion. Requirement U10 is formulated ambiguously. The word "maintain" could mean that the system is *not allowed* to change the state of the luminaries as long as the room is occupied. More probably, however, it is intended to mean that the system is *required* to change the state of the luminaries when, e.g., the illumination measured by the outdoor light sensor changes.

The requirement refers to the *current* light scene, whereas Requirement U5 (as well as U1–U3) talks about the *chosen* light scene. We have decided to use the same function $light\_scene$ in both formalizations. The function $light\_scene$ is also used in the formalization of Requirement U3. It should be discussed with the customers if in all four requirements the same function can be used or not. We intend the function $light\_scene$ to yield the light scene that is *required* to be established in a given state. Details of the functions $light\_scene$, $light\_state$, and the predicate $conforms$ must be defined in the specification phase.

Our formalization states that the state of the luminaries must *always* satisfy the ambient light level specified in the required light scene. In reality, this might not be possible, for example when luminaries are broken. Hence, we have realized that requirements are probably *missing*. In Section 3.2.1 of the problem description, where fault tolerance is discussed, only sensor failures are taken into account.

From this discussion, it follows that Requirement FM1 (use daylight to achieve the desired light setting of each room whenever possible) is already taken into account. Light scenes are defined as follows: "a light scene is a predefined setting of the ambient light level and a prescription that determines in which way the ceiling light groups should be used to achieve the ambient light level". This definition seems to imply that the luminaries are switched on only if the daylight is not sufficient. It would be more appropriate, however, to talk about using daylight explicitly in the description of light scenes.

Requirement FM3. The ceiling light groups in a room have to be off when the room is unoccupied for at least $T3$ minutes.
This requirement is similar to U3. Its formalization is:

$$\begin{array}{l}\forall\, tr : Tr \bullet \forall\, i, j : \mathsf{dom}\, tr \mid tr(j).t - tr(i).t \geq T3 \bullet \\ \quad tr(i).e = leave\_last \wedge (\forall\, k : i+1 \mathrel{..} j \bullet \neg\, occupied(tr(k).s)) \\ \qquad \Rightarrow (\exists\, l : i+1 \mathrel{..} j \bullet tr(l).e = turn\_off \wedge tr(l).t = tr(i).t + T3)\end{array}$$

Requirement FM5 belongs to the transformational part of the system, as well as requirements FM8, FM9, FM10, and FM11.

Requirement FM6. The facility manager can turn off the ceiling light groups in a room that is not occupied.
Formalization:

$$\begin{array}{l}\forall\, tr : Tr \mid \neg\, occupied(last(tr).s) \bullet \exists\, tr' : Tr \bullet front\, tr' = front\, tr \wedge \\ \quad tr'(\#tr).s = tr(\#tr).s \wedge tr'(\#tr).e = turn\_off\_manual\end{array}$$

This formula says that a trace that ends with a state in which the room is not occupied can be continued with an event $turn\_off\_manual$.

Schematic expression: $\neg\, occupied \leadsto turn\_off\_manual$

Semantic relations: none.

Interaction candidates: they are U3, U10, and FM3, because of the common preconditions $\neg\, occupied$ and $true$. There could possibly be an interaction with FM3. If $turn\_off$ has already occurred, is it possible or allowed for $turn\_off\_manual$ to occur and vice versa? If not both events are allowed, FM3 and FM6 could be replaced by the following constraint which is a weakening of FM3: either the system turns off the light after $T3$ minutes, or the facility manager turns off the light before $T3$ minutes have passed:

$$\begin{array}{l}\forall\, tr : Tr \bullet \forall\, i, j : \mathsf{dom}\, tr \mid tr(j).t - tr(i).t \geq T3 \bullet \\ \quad tr(i).e = leave\_last \wedge (\forall\, k : i+1 \mathrel{..} j \bullet \neg\, occupied(tr(k).s)) \\ \qquad \Rightarrow (\exists\, l : i+1 \mathrel{..} j \bullet \mathbf{tr(l).e = turn\_off\_manual} \wedge \mathbf{tr(l).t < tr(i).t + T3} \\ \qquad\quad \vee\, tr(l).e = turn\_off \wedge tr(l).t = tr(i).t + T3)\end{array}$$

Discussion. First, it is not stated how the facility manager can turn off the lights. Probably this is achieved using the facility manager control panel.

Second, the relation between FM3 and FM6 should be discussed with the customers.

Requirement FM7. If a malfunction occurs, the facility manager has to be informed. Formalization:

$$\forall\, d : DEVICE \;\bullet\; fail(d) \;\; immediately\_followed\_by \;\; inform\_facility\_manager(d)$$

Schematic expression: $fail \leadsto inform\_facility\_manager$

Semantic relations: none.

Interaction candidates: they are U8 and U10 because of the common preconditions $fail$ and $true$.

We judge that there is indeed an interaction with U8. Although it is not stated explicitly, it seems reasonable to inform not only the user but also the facility manager in case of a failure of the outdoor light sensor or the motion detector. Hence, we revise the formalization of U8 as follows ($followed\_by$ is defined in Appendix A):

$$\begin{aligned}&\forall\, d : DEVICE \;\bullet\; outdoor\_light\_sensor(d) \vee motion\_detector(d)\\ &\quad \Rightarrow fail(d) \;\; followed\_by \;\; \{inform\_user(d), \mathbf{inform\_facility\_manager(d)}\}\end{aligned}$$

The informal requirement U8 has to be updated. If any outdoor light sensor or the motion detector of a room does not work correctly, the user of this room and the facility manager have to be informed.

Discussion. We have decided to use the same event $fail$ to model what is called "malfunction" and what is called "failure" in the problem description. These two terms seem to be equivalent.

Note that the handling of $fail$ events must be implemented in such a way that the failure is stored, as required by Requirement FM10. This must be specified formally in the specification phase.

This section has illustrated how formalization of requirements and interaction analysis help in detecting flaws in informal requirements documents.

4.3 A Summary of Produced Documents

The result of our work consists of several documents:

1. a diagram to fix the domain vocabulary and to express *static* aspects of the system (see Figure 4);
2. a list of events, together with their classification, concerning the *reactive* part of the system (see results of Step 3);
3. a list of system operations, concerning the *transformational* part of the system (see Table 1);

4. an updated informal requirements document;
5. a list of functions and predicates on the system state that were introduced when formalizing the requirements. We have defined the following predicates on the system state and a room: $occupied$, $safe\_illumination$, $conforms(light\_state, light\_scene)$, $lesseqT1$, $same\_light\_scene$ and $select\_light\_scene\_established$.
   The functions defined on the system state are:
   - $light\_scene : STATE \rightarrow LIGHT\_SCENE$
   - $light\_state : STATE \rightarrow LIGHT\_STATE$
6. tables of semantic relations between event and predicate symbols, see Appendix B.2;
7. a formalization of the requirements, expressed as constraints on admissible system event traces;
8. schematic expressions for the requirements, together with interaction candidates, see Appendix B.1.

5 Discussion of the Method and its Application to the Case Study

Performing this case study served us to further validate our requirements engineering method. We have used our method on several case studies, including an elevator system [HS99b], an automatic teller machine [HS99a], and an access control system [SH00].

5.1 Results Concerning the Informal Requirements Document

Using our systematic approach, we found several problems with the informal requirements document that were discussed in Section 4.2. In summary, these problems concern the following aspects:

Incoherent vocabulary.

- We could not find out the difference between the *current* and the *chosen* light scene. This difference should be explained in the dictionary.
- The difference between a *light scene* and a *light setting* is not clear for us. In the dictionary, the *desired light setting* is explained as a "setting of a ceiling light group". A setting could be what was called *light state* in Section 4.2, U10. However, this seems to contradict the usage of the word "setting" in the definition of a light scene, where a light scene is a "predefined setting of the ambient light level".
- It seems to us that *failure* and *malfunction* are the same thing, but this does not become clear from the text.

Unused details. In paragraph 2.8 of the problem description, *door closed contacts* are mentioned. However, the states of these contacts are not referred to in the requirements. It is not clear to us what the contacts are used for. If they are not used at all, they should not be installed. Otherwise, they should occur in the requirements.

Organization of the requirements document. Requirement U9 is of a different nature than (most of) the other requirements. It concerns the hardware to be purchased, whereas the other requirements concern the behavior of the light control system. We propose not to mix these different kinds of requirements, but to state the requirements concerning the hardware in a separate section.

Ambiguous requirements. We have identified the requirements U1, U2, and U10 as ambiguous. These requirements should be further discussed with the customers.

Unsatisfiable requirements. It is not clear to us how Requirements U1 and U10 can be satisfied, i.e., how it can be guaranteed that the illumination is always safe and that the state of the light groups always conforms to the current light scene.

Incoherent requirements. Our algorithm points out pairs of requirements that might interact. For the case study, we see possible interactions between U1 and U5, U1 and U10, and FM3 and FM6.

Missing requirements. A number of requirements should be stated in addition to the ones in the problem description document:

- If parts of the system fail, how can it be brought back to normal functionality?
- What is the effect of pushing a button on the chosen and current light scene, respectively?
- *How* is the daylight used to achieve the desired light setting or to establish a light scene?
- How are users and the facility manager informed when a failure occurs?
- How are failures detected?
- What happens when luminaries are broken?

Redundant requirements. Requirements U2 and U10 seem to be similar. They should either be joined or reformulated to make the difference clear.

5.2 Results Concerning the Method

Requirements define a set of conditions that must be met by a system or a system component to satisfy a contract, standard or other imposed document or description. For example, the IEEE Standard 1498 [IEE94] defines a requirement as a characteristic that a system or a software item must possess in order to be acceptable to the acquirer. Requirements should be stated completely and unambiguously. Our method helps to detect missing, incoherent, and unsatisfiable requirements, as was demonstrated in Sections 4 and 5.1. Formalizing the requirements supports eliminating ambiguities.

Furthermore, formal requirements have another advantage: it is possible to define a notion of *correctness* of a specification with respect to the requirements, see [HS99b].

Requirements traceability is an important issue in requirements engineering. Jarke [Jar98] defines requirements traceability as the ability to describe and follow the life of a requirement, in both a forward and backward direction. In our method, traceability is guaranteed in the following way:

– Single requirements are fragments. The smaller the requirements, the better traceable they are, because their realization does not distribute over large parts of the system.
– For each event and each predicate that is introduced, it is noted in which requirements it is used.
– For each part of the formal specification, we can name the requirements that lead us to define it in the way we did.

Performing the case study has shown that our method is well suited to achieve the goals stated in Section 1. As could be expected, the different steps of the method were not performed separately and in exactly the given order. Moreover, several repetitions of the various steps were necessary. Nevertheless, it is important to define separate steps, because they structure the requirements elicitation process and help focus attention to crucial points [PC86].

In summary, our approach gives substantial methodological guidance for requirements elicitation, without introducing a new language or a new formalism. The requirements elicitation phase is independent of the specification language to be used later. We propose a standardized way of expressing facts, assumptions and requirements. Constraints on the set of possible traces are a very flexible and powerful means of describing a system and its interaction with the environment. Expressing requirements as constraints on traces makes it possible to systematically detect conflicting requirements and to define a formal notion of correctness of a specification with respect to a set of requirements.

References

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[CAB+94] D. Coleman, P. Arnold, St. Bodoff, Ch. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[DL96] R. Darimont and A.van Lamsweerde. Formal Refinement for Patterns for Goal-Driven Requirements Elaboration. In *Proc FSE-4, ACM Symposium on the Foundation of Software Engineering*, pages 179–190, 1996.

[EN96] S. Easterbrook and B. Nuseibeh. Using ViewPoints for inconsistency management. *Software Engineering Journal*, pages 31–43, January 1996.

[FFH91] M.S. Feather, S. Fickas, and R.B. Helm. Composite System Design : the Good News and the Bad News. In *Proc. 6th Knowledge-Based Software Engineering Conference*, pages 16–25. IEEE Computer Society Press, 1991.

[HS98] M. Heisel and J. Souquières. A heuristic approach to detect feature interactions in requirements. In K. Kimbler and W. Bouma, editors, *Proc. 5th Feature Interaction Workshop*, pages 165–171. IOS Press Amsterdam, 1998.

[HS99a] M. Heisel and J. Souquières. A method for requirements elicitation and formal specification. In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, *Proceedings 18th International Conference on Conceptual Modeling, ER'99*, LNCS 1728, pages 309–324. Springer-Verlag, 1999.

[HS99b] M. Heisel and J. Souquières. De l'élicitation des besoins à la spécification formelle. *Technique et science informatiques*, 18(7):777—801, 1999.

[IEE94] IEEE94. Software development. IEEE publications office, IEEE Standard 1498, Los Alamitos, CA, March 1994.

[Jar98] M. Jarke. Requirements tracing. *Communications of the ACM*, pages 32–36, December 1998.

[JZ95] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[Par95] D.L. Parnas. Using Mathematical Models in the Inspection of Critical Systems. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*, pages 17–31. Prentice Hall, 1995.

[PC86] D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. *IEEE Transactions on Software Engineering*, 12(2), 1986.

[SH00] J. Souquières and M. Heisel. Une méthode pour l'élicitation des besoins : application au système de contrôle d'accès. In Yves Ledru, editor, *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels, AFADL'2000*, 2000. To appear.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd edition, 1992.

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

A Formal Expression of Constraints on Traces

In the following specification of system traces, we use the Z notation [Spi92].

$$[STATE, EVENT, TIME]$$

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{TraceItem}} \\ \hline s : STATE \\ e : EVENT \\ t : TIME \\ \hline \end{array}$$

Each trace of the system is a sequence of trace items, where events later in the sequence must not happen earlier in time than events earlier in the sequence. The sign $\leq_t$ denotes a relation "not later" on time, which fulfills the axioms of a partial ordering relation.

For each system, we will call the set of admissible traces Tr. Constraints will be expressed as formulas restricting the set Tr. For each possible trace, its prefixes are also possible traces.

$$\begin{array}{|l} TRACE : \mathbb{P}(\mathrm{seq}\ TraceItem) \\ \hline \forall\, tr : TRACE \bullet \forall\, i : \mathrm{dom}\, tr \bullet i = \#tr \lor \\ \qquad (tr\ i).t \leq_t (tr(i+1)).t \end{array}$$

$$\begin{array}{|l} Tr : \mathbb{P}\, TRACE \\ \hline \forall\, tr : Tr;\ tr' : TRACE \mid \\ \qquad tr'\ \mathrm{prefix}\ tr \bullet tr' \in Tr \end{array}$$

To express constraints concisely, we define two specification macros. The first one says that event e_1 is immediately followed by e_2 in all traces where e_1 is not the last event. The second one says that event ev is followed by a set of events es, where the order in which the events in es occur is insignificant.

$$\begin{array}{|l} \_immediately\_followed\_by\_ : EVENT \leftrightarrow EVENT \\ \hline \forall\, e_1, e_2 : EVENT \bullet e_1\ immediately\_followed\_by\ e_2 \\ \qquad \Leftrightarrow \forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \mid tr(i).e = e_1 \bullet i = \#tr \lor tr(i+1).e = e_2) \end{array}$$

$$
\begin{array}{|l}
\_followed\_by\_ : EVENT \leftrightarrow \mathbb{F}\, EVENT \\
\hline
\forall\, ev : EVENT;\ es : \mathbb{F}\, EVENT \bullet ev\ followed\_by\ es \\
\quad \Leftrightarrow (\forall\, tr_1, tr_2 : Tr \mid (last\ tr_1).e = ev \land tr_1 \text{ prefix } tr_2 \land \#tr_2 - \#tr_1 \geq \#es \bullet \\
\qquad \{i : \#tr_1 + 1 \mathrel{..} \#tr_1 + 1 + \#es \bullet tr_2(i).e\} = es)
\end{array}
$$

B Results of the Interaction Analysis

B.1 Schematic Constraints and Interaction Candidates

Req.	schematic expressions	Candidates	Interactions
U1	$occupied \leadsto safe\_illumination$	–	–
U2	$occupied \leadsto \neg\ turn\_off$	U1	–
U3	$leave\_last \land enter\_first \land \neg\ occupied$ $\land\ lesseqT1 \leadsto same\_light\_scene$	U1, U2	–
U5	$select\_light\_scene \leadsto$ $selected\_light\_scene\_established$	U1, U2, U3	U1?
U8	$fail \leadsto inform\_user \land$ $\mathbf{inform\_facility\_manager}$	–	–
U10	$true \leadsto conforms$	U1–U5, U8	U1?
FM3	$leave\_last \land \neg\ occupied \leadsto turn\_off$	U1–U5, U10	–
FM6	$\neg\ occupied \leadsto turn\_off\_manual$	U3, U10, FM3	FM3?
FM7	$fail \leadsto inform\_facility\_manager$	U8, U10	U8

B.2 Semantic Tables

Necessary conditions for events

$\neg\ occupied \leftsquigarrow enter\_first$ $\quad$ $occupied \leftsquigarrow press$

$occupied \leftsquigarrow leave\_last$ $\quad$ $occupied \leftsquigarrow select\_light\_scene$

Events establishing predicate literals

$$
\begin{aligned}
enter\_first &\leadsto occupied \\
leave\_last &\leadsto \neg\ occupied \\
enter\_first &\leadsto safe\_illumination \\
turn\_off &\leadsto \neg\ safe\_illumination \\
turn\_off\_manual &\leadsto \neg\ safe\_illumination \\
enter\_first &\leadsto same\_light\_scene \\
select\_light\_scene &\leadsto \neg\ same\_light\_scene \\
select\_light\_scene &\leadsto \neg\ default\_light\_scene \\
select\_light\_scene &\leadsto selected\_light\_scene\_established \\
turn\_off &\leadsto \neg\ selected\_light\_scene\_established \\
turn\_off\_manual &\leadsto \neg\ selected\_light\_scene\_established
\end{aligned}
$$

Relations between predicate literals

$occupied_{\Rightarrow} = \{safe\_illumination\}$