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

Jeanine Souquières
LORIA—Université Nancy2
B.P. 239 Bâtiment LORIA
F-54506 Vandœuvre-les-Nancy, France
Jeanine.Souquieres@loria.fr

Maritta Heisel
Fakultät für Informatik
Universität Magdeburg
D-39016 Magdeburg, Germany
heisel@cs.uni-magdeburg.de

December 16, 1999

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

1 Motivation

Eliciting requirements for a software system is a complex conceptual and practical activity. In particular, communication between customers and software analysts often is not perfect, which leads to misunderstandings. Customers may forget or may not be aware of requirements, and analysts, without the deep domain knowledge of most customers, may not be able to fill in the gaps. In addition, some requirements and constraints may be so obvious from an expert customer point of view that they do not seem worth mentioning. But the lack of domain expertise among analyst teams means that these unstated requirements may not be recognized and hence are not incorporated in the development process. Customers seldom fully understand their requirements at the beginning of the project. They often lack a clear vision of what the system should do and change their minds during development. In general, they only learn more about what they want when they are delivered a software system that does *not* exhibit the desired features [AP98].

Giving methodological guidelines for the first steps of the software development [DvLF93] is very important to increase the reliability of software and to decrease its costs. Analysis errors are the most numerous, the most persistent, and the most dangerous ones for the rest of the development lifecycle. They are the most expensive to detect and correct: an analysis error detected during design is five times as expensive as if it had been discovered during the analysis process; this ratio is about two hundred if it is detected only during the operation stage of the software. These results, published by the Standish Group[1], are issued from recent inquiries about success and failure of American computer science projects.

Because it is crucial for the success of software projects that the requirements be correct, unambigious and complete, a systematic approach to requirements elicitation that helps detect incoherences, incompletnesses and ambiguities is of great value.

[1]They can be found at the Internet address *http://www.standishgroup.com/chaos.html*.

We have developed a method to perform the early phases of the software lifecycle in a systematic way˜citeHeiselSouquieres99b,HeiselSouquieres99a. The method supports two phases, nameley requirements elicitation and specification development. In this paper, we describe the requirements elicitation phase and apply it to the light control case study.

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,
- find missing requirements,
- establish an adequate starting point for a formal specification of the software system.

We integrate some formalization very early in the development in order to analyze the customer needs in a detailed way and to reveal problems by means of the encountered difficulties during the formalization process. In particular, an analysis of possible interactions between requirements is performed all along this formalization process. Requirements should be stated as *fragments* as small as possible. Such fragments can correspond to parts of independent scenarios of system behavior. For reasons of traceability, each fragment should be given a unique name or number. 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 in the form of an entity-relationship diagram, and several formal documents.

Section 2 presents 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 exhibited by its application to the case study.

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], OMT [RBP+91], or UML [FS97]. 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, consistency analyses are performed. Their purpose is to obtain a consistent set of requirements. The method for requirements elicitation (and also for specification development) is described in [HS99a], the method to detect interactions in requirements 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. *Facts* express phenomena that always hold in the application domain, regardless of the software system implementation. These are often hardware properties, e.g., that a card reader can hold at most one card. Other requirements cannot be enforced because, e.g., human users might violate regulations. In such a case, it may be possible that the system can fulfill its purpose properly only under the condition that users behave as required. Such conditions are expressed as *assumptions*, which can be used to show that the software system is correctly implemented. *Requirements* constrain the specification and implementation of the software system. It must be demonstrated that they are fulfilled, once the specification and the implementation are finished. In the specification phase, it must be shown that the modeling of the system takes the facts into account and that the requirements are fulfilled, whereas assumptions can be taken for granted.

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 is 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 is concerned with 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 other constraints should be analyzed, in order to detect inconsistencies or undesired behaviors [HS98]. This step is itself decomposed into six steps described below.

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 :

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

Let be *Tr* the set of possible traces. Constraints will be expressed as formulas 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 e occurs. For each possible trace, its prefixes are also possible traces. A formal specification of traces is defined in the 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 will 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 must or may occur", 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*. The set of predicate literals is denoted *PLit*. *Event literals* are defined analogously. For formal expressions, we use the syntax of the specification language Z [Spi92].

In order to systematically express formal requirements and to facilitate interaction analysis, 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 (invariants).

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

$$\forall\, tr : Tr;\ room : ROOM \bullet \forall\, i : dom\, tr \bullet occupied(tr(i).s, room) \Rightarrow safe_illumination(tr(i).s, room)$$

The variable tr denotes an admissible system trace, and the variable i denotes the i-th element of the trace, consisting of a state, an event, and a time. The expression $dom\, tr$ denotes the valid indices of the trace, i.e., $1 \mathinner{\ldotp\ldotp} \#tr$, where $\#$ denotes the length of a trace.

Step 5.2 : Give a schematic expression of the constraint. We have defined an algorithm that computes 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 following form :

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

where the x_i, y_j are literals (event symbols, predicates or their negations) and $\diamond$ denotes either conjunction or disjunction. The $\leadsto$ symbol indicates that the precondition refers to an earlier state than the postcondition. If the constraint is an invariant of the system state, it is replaced by $\Rightarrow$.

For transforming a constraint into its schematic form, we abstract from quantifiers and from parameters of predicate and event symbols.

Example. If the last person hast left the roon and the room remains unoccupied for some time, the lights must be switched off (cf. FM3): $leave_last \wedge \neg\, occupied \leadsto turn_off$

Step 5.3 : Update tables of semantic relations. The detection of constraint interactions cannot be based on the 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. These state which predicates on the system state have to be true in order for the event to occur. If the event e can only occur when 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 : $\neg\, p_{\Rightarrow} = \{q : PLit \mid \neg\, p \Rightarrow q\}$
   - the set of predicate literals that entail it: ${}_{\Rightarrow}p = \{q : PLit \mid q \Rightarrow p\}$
   - the set of predicate literals that entail its negation: ${}_{\Rightarrow}\neg\, p = \{q : PLit \mid q \Rightarrow \neg\, p\}$

   Example. $occupied_{\Rightarrow} = \{safe_illumination\}$

   By contraposition, the following equalities hold:

   ${}_{\Rightarrow}\neg\, p = \{pl : p_{\Rightarrow} \bullet \neg\, pl\}$
   ${}_{\Rightarrow}p = \{pl : \neg\, p_{\Rightarrow} \bullet \neg\, pl\}$

   Hence, only two of the four sets must be determined explicitely.

   These tables are not only useful to detect interactions; they are also useful to develop and validate the 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 semantic relation tables constructed in Step 5.3, using an automatic procedure presented. In general, two constraints are interaction candidates for one another if they have common preconditions but incompatible postconditions, as is illustrated in Figure 1.

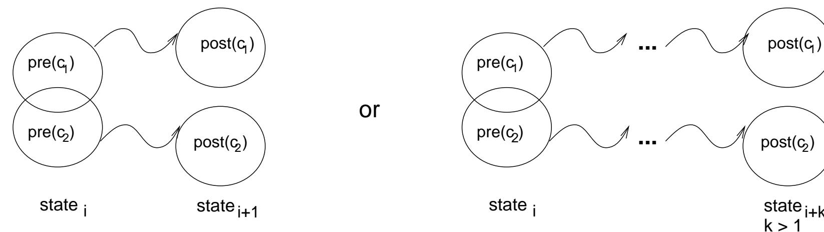

Figure 1: Interaction candidates

The left hand-side of the figure shows the situation where the incompatibility of postconditions manifests itself in the state immediately following the state that is referred to by the precondition. The right-hand side shows that the incompatibility may also occur in a later state.

Our method to determine interaction candidates consists of two parts : precondition interaction analysis determines constraints with preconditions that are neither exclusive nor independent of each other. This means, there are situations where both constraints might apply. Their postconditions 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. To decide if two constraints $\underline{x} \leadsto \underline{y}$[2] and $\underline{u} \leadsto \underline{w}$ might interact on their preconditions, we perform the following reasoning: if the two constraints have common literals in their precondition (i.e., $\underline{x} \cap \underline{u} \neq \varnothing$), then they are certainly interaction candidates.

But the common precondition may also be hidden. For example, if $\underline{x}$ contains the event e, and $\underline{u}$ contains the predicate literal p, where e is only possible if p holds ($p \leftsquigarrow e$), then we also have detected a common precondition between the two constraints.

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

Formally, the set C_{pre} of precondition interaction candidates of a new constraint c' with respect to an already existing set *far* of facts, assumptions, and requirements is defined as follows. Figure 2 illustrates the definition.

$$
\begin{aligned}
&C_{pre}(c', far) = \\
&\quad \{c : far \mid precond(c) \cap precond(c') \neq \varnothing\} \\
&\quad \cup \\
&\quad \textstyle\bigcup_{x \in pre_predicates(c')} \{c : far \mid ((_{\Rightarrow}x \cup x_{\Rightarrow}) \cap precond(c) \neq \varnothing) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\exists\, e : precond(c) \cap EVENT;\ y : {}_{\Rightarrow}x \cup x_{\Rightarrow} \bullet y \leftsquigarrow e)\}
\end{aligned}
$$

where

$$
\begin{aligned}
&{}_{\leadsto}e = \{pl : PLit \mid pl \leadsto e\} \\
&pre_predicates(c) = (precond(c) \cap PLit) \cup \textstyle\bigcup_{e \in precond(c) \cap EVENT} {}_{\leadsto}e
\end{aligned}
$$

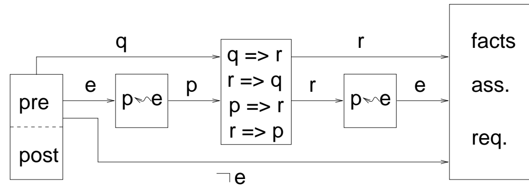

Figure 2: Determining interaction candidates by precondition analysis

Postcondition Interaction Analysis. For determining candidates for postcondition interaction, we proceed similarly. To find conflicting postconditions, we perform forward chaining on the postconditions of the new constraint, negate the resulting literals, and check if one of the negated literals follows from the postcondition of another constraint. This constraint is then identified as an interaction candidate. To perform forward chaining on events, the information contained in the table of events establishing predicate literals ($e \leadsto p$) is used. As in the definition of C_{pre}, no chaining is performed on negative event literals. Figure 3 illustrates the formal definition of the function C_{post}.

[2]Underlined identifiers denote sets of literals.

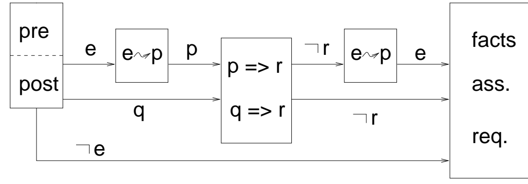

Figure 3: Determining interaction candidates by postcondition analysis

$$
\begin{aligned}
&C_{post}(c', far) = \\
&\quad \{c : far \mid postcond(c)\ opposite\ postcond(c')\} \\
&\quad \cup \\
&\quad \{c : far \mid \exists\, x : post_predicates(c);\ \ y : post_predicates(c') \bullet x_{\Rightarrow}\ opposite\ y_{\Rightarrow}\}
\end{aligned}
$$

where

$$
\begin{aligned}
&e_{\leadsto} = \{pl : PLit \mid e \leadsto pl\} \\
&post_predicates(c) = (postcond(c) \cap PLit) \cup \bigcup_{e \in postcond(c) \cap EVENT} e_{\leadsto} \\
&ls_1\ opposite\ ls_2 \Leftrightarrow \exists\, x : ls_1 \bullet \neg\, x \in ls_2
\end{aligned}
$$

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

It must be noted that this approach for the detection of feature interactions is *heuristic*. This means, we cannot guarantee that all interactions that might occur are found by our automatic procedure.

Step 5.5 : Decide if there are interactions of the new constraint with the candidates determined in Step 5.4. It is up to the analysts and customers to decide if the conjunction of the new constraint with the candidates yield an unwanted behavior or not and how detected interactions can be resolved.

Step 5.6 : Take into account each interaction. If an interaction occurs, take one of the following actions : (i) correct a fact corresponding to the formalization of the domain knowledge, (ii) relax a requirement (by adding a new pre- or postcondition, as preconditions are usually conjunctions, and postconditions are usually disjunctions) or (iii) strengthen an assumption. Once a constraint has been modified, perform an interaction analysis on those literals that were changed or newly introduced.

Validation conditions associated with Step 5 are the following :

- each requirement of Step 2 must be expressed,
- the set of constraints must be consistent,
- for each introduced predicate, events that modify it must be shared with the software system.

Steps 5.1 to 5.6 allow mutual coherence between the set of constraints to be preserved. Usually, revisions and interactions with customers will be necessary.

3 Related work

Before we apply our method to the light control case study, we discuss related work.

Separating domain knowledge from requirements and checking consistency between these 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 the requirements of the computer system. 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 these quantities by mathematical variables. He proposes to characterize environmental quantities as either monitored or controlled. This corresponds to the classification of events in our and Jackson and Zave's approach [JZ95]. Reading a monitored quantity corresponds to an event observable by the software system, and controlling a quantity corresponds to events controlled by the software system.

Whereas assumptions are not used by Parnas and by Jackson and Zave, they do play a role in KAOS [DLF93, 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, starting with the elaboration of the goals in an AND/OR structure and ending with the assignment of responsibilities to agents. KAOS is similar to our approach in that it provides heuristics for requirements elicitation and specification development. It is distinguished from our approach in that it uses its own language and in that it 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.

We have already referred to the work of Jackson and Zave [JZ95, ZJ97] several times. But although our method is inspired by their approach to requirements engineering, there are some important differences: first, Jackson and Zave consider a specification to be a special kind of requirement. Requirements (and thus specifications) 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, in our approach, 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. These issues are not addressed explicitly by Jackson and Zave.

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 the Light Control Case Study

We now apply the method described in Section 2 to the light control system, carrying out the different steps one by one. The starting point of the requirements elicitation process is the problem description made available to all authors of this special issue. In this paper, we only treat the requirements concerned with offices. The requirements for hallways, laboratories, and other rooms can be treated analogously. We use the numbering of requirements as given in the problem description.

4.1 Informal Steps

Step 1 : Introduce the domain vocabulary. The domain vocabulary for this case study is given in Table 3 of the problem description. Figure 4 contains those notions we actually used in carrying out the part of the case study concerned with offices. This figure summarizes the results of Steps 1–4 of our method.

Step 2 : State the facts, assumptions, and requirements. The problem description is quite well structured with a good fragmentation of the different requirements, each of them being numbered.

Facts correspond to the descriptions given in Part 2 of the document. There are no assumptions on user behavior. In the following, we concentrate on the requirements described in Part 3 of the problem description.

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 are :
  - *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 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 are :
  - *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	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 development phase, and hence are not given in this paper.

An entity-relationship like diagram gives a visual representation of the terminology used in the first informal steps of our method, see Figure 4. 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.

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 offices. For each formalized requirement, we discuss unclear points and decisions taken. A complete summary of the results of Step 5 can be found in Appendix B.

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 :

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

To formally express U1, two predicates on the system state have been introduced, *occupied* and *safe_illumination*, both additionally parameterized with a room[3].

Schematic expression : $occupied \Rightarrow safe_illumination$

Semantic relations :
First, we specify which events modify the two new predicates:
$enter_first \leadsto occupied$
$leave_last \leadsto \neg\, occupied$
$enter_first \leadsto safe_illumination$

[3]To enhance readability of the constraints, we only state the requirements for single rooms in the following, i.e., we do not mention rooms as parameters any more.

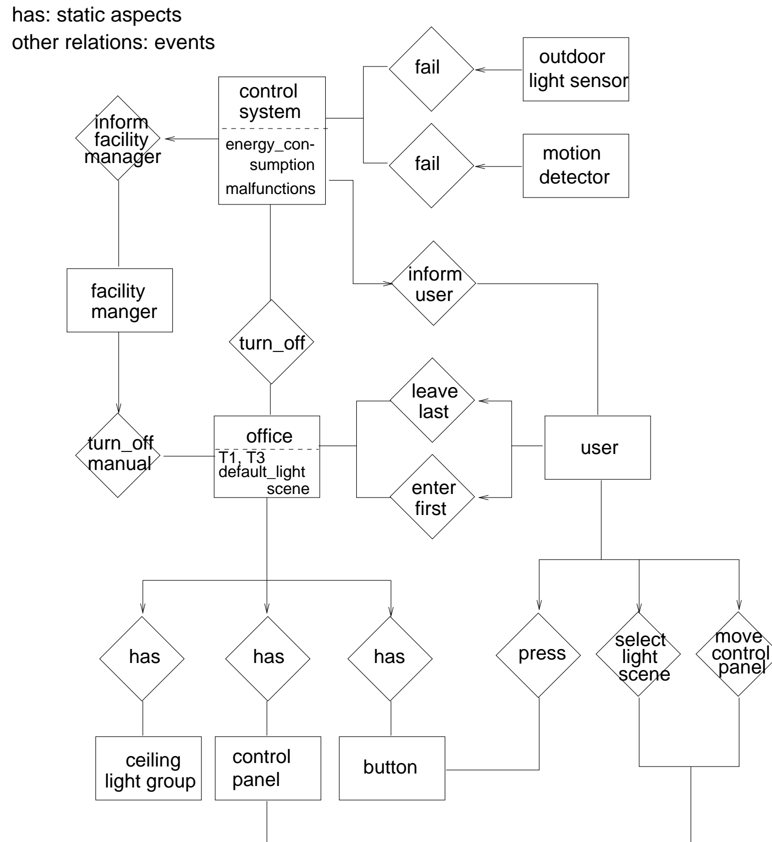

Figure 4: Domain vocabulary for the light control system

$select_light_scene \leadsto safe_illumination$ veut-on vraiment avoir ca?
$turn_off \leadsto \neg\ safe_illumination$
$turn_off_manual \leadsto \neg\ safe_illumination$

Second, the predicate *occupied* and its negation are necessary conditions for some events:
$\neg\ occupied \leftsquigarrow enter_first$
$occupied \leftsquigarrow leave_last$
$occupied \leftsquigarrow press$
$occupied \leftsquigarrow select_light_scene$

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 *ambigous requirement*. In a real project, custumors 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 office, 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 automatic procedure will detect inconsistencies of this formalization with other requirements, which should cause us to weaken the constraint.

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 : $occupied \leadsto \neg\, turn_off$

Semantic relations : none, because no new symbols have been introduced

Interaction candidates:
U1 and U2 have the same precondition but their postconditions do no exclude each other.

Discussion :
Requirement U2 is ambigous, 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 chosen to intrepret U2 meaning that *the system* is not allowed to change the light scene established in the office. 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.

A similar problem occurs with Requirement U10 (see the discussion there).

Requirement 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.
Formalization : To formalize U3, we introduce a function

$light_scene : STATE \rightarrow LIGHT_SCENE$

which denotes the light scene required for a room in a given state.

$$\begin{aligned} &\forall\, tr : Tr \bullet \forall\, i,j : \mathsf{dom}\, tr \mid j > i \wedge j+1 \in \mathsf{dom}\, tr \bullet \\ &\quad tr(i).e = leave_last \wedge tr(j).e = enter_first \\ &\quad \wedge (\forall\, k : i \,..\, j \bullet \neg\, occupied(tr(k).s)) \wedge tr(j).t - tr(i).t \leq T1 \\ &\qquad \Rightarrow light_scene(tr(j+1).s) = light_scene(tr(i).s) \end{aligned}$$

The premise of the formula captures the fact that the interval in which the room is not occupied is marked by the two 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 : $leave_last \wedge enter_first \wedge \neg\, occupied \wedge lesseqT1 \leadsto same_light_scene$

Two new predicates have been introduced, $lesseqT1$ and *same_light_scene* to avoid functions in schematic expressions.
Semantic relations [4]:
It is clear that we should have the following relations:
$enter_first \leadsto same_light_scene$

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

$select_light_scene \rightsquigarrow \neg\ same_light_scene$
But what about:
$press \rightsquigarrow \neg\ same_light_scene$
$turn_off \rightsquigarrow \neg\ same_light_scene$
$turn_off_manual \rightsquigarrow \neg\ same_light_scene$?
These are again questions to discuss with the customers.

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 entries into the semantic tables are unclear.

Requirement 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.

U4 is formalized in the same manner as U3, using a function *default_light_scene* that yields the default light scene that is set for an office in a given state. For details, see Appendix B.

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 : $select_light_scene \rightsquigarrow selected_light_scene_established$

Semantic relations:
$select_light_scene \rightsquigarrow selected_light_scene_established$
$turn_off \rightsquigarrow \neg\ selected_light_scene_established$

Interaction candidates: Because the event *select_light_scene* is only possible if the predicate *occupied* holds, U1–U4 are interaction candidates. We judge that there is no interaction with U2–U4, 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, discussing with the customers.

Discussion :
Apart from the possible interaction with U1 and its resolution, there are no further unclear points in this requirement.

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 understand 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 light control system.

In contrast, the default light scene and the value T1 can be regarded as attributes of an office. Hence, U6 and U7 concern the transformational part of the light control system. In Step 4 of our method, we have introduced the system operations *set_default_light_scene* and *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]. This phase is not treated in the present paper.

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 :

$\forall\, device : \{outdoor_light_sensor, motion_detector\} \bullet$
$\quad fail(device)\ immediately_followed_by\ inform_user(device)$

This formalization uses a type *DEVICE* with elements *outdoor_light_sensor* and *motion_detector*[5].

Schematic expression : $fail \leadsto inform_user$

Semantic relations: none.

Interaction candidates: none.

Discussion :
The requirement and its formalization say 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: the hardware requirements should be stated in a separate section.

Requirement U10. The ceiling light groups should be maintained by the control system depending of 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* on the system 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.

[5]For the definition of *immediately_followed_by*, see Appendix A.

Schematic expression : $\mathit{true} \Rightarrow \mathit{conforms}$

Semantic relations: none.
Normally, we should give events that establish and events that falsify the predicate *conforms*. However, we require *conforms* to be an invariant of the system. Hence, there are no events that falsify it (but see discussion of this requirement). Also there 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 $p \Rightarrow \mathit{true}$ for all predicates p, we get as interaction candidates all requirements formalized so far, i.e., U1–U5, U8. As for U5, there might be an interaction with U1. How exactly can safe illumination be guaranteed?

Discussion :
This requirement if 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 formalizations of Requirements U3 and U4. 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 of our method.

Our formalization says 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*. Note that in Section 3.2.1 of the problem description that discusses fault tolerance, 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. In the definition of light scenes it is implicitly stated that the luminaries are switched on only if the daylight is not sufficient. It would be more appropriate, however, to talk about using daylight explicitely 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 and U4. Hence, we do not discuss it in detail. Its formalization is:

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

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 :

$$\forall\, tr : Tr \mid \neg\, occupied(tr(\#tr).s) \bullet$$
$$\quad \exists\, tr' : Tr \bullet front\, tr' = front\, tr \wedge tr'(\#tr).s = tr(\#tr).s \wedge tr'(\#tr).e = turn_off_manual$$

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*. Hence, when the room is not occupied, it is possible for the event *turn_off_manual* to occur.

Schematic expression : $\neg\, occupied \leadsto turn_off_manual$

Semantic relations: none.

Interaction candidates: U3, U4, U10, FM3.
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 (a weakening of FM3), which expresses that either the system turns off the light after T3 minutes, or the facility manager turns off the light before T3 minutes have passed:

$$\forall\, tr : Tr \bullet \forall\, i, j : \mathrm{dom}\, tr \mid tr(j).t - tr(i).t \geq T3 \bullet$$
$$\quad tr(i).e = leave_last \wedge (\forall\, k : i+1 \mathinner{..} j \bullet \neg\, occupied(tr(k).s))$$
$$\quad\quad \Rightarrow (\exists\, l : i+1 \mathinner{..} j \bullet$$
$$\quad\quad\quad tr(l).e = turn_off_manual \wedge tr(l).t < tr(i).t + T3$$
$$\quad\quad\quad \vee\, tr(l).e = turn_off \wedge tr(l).t = tr(i).t + T3)$$

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: U8, U10.
There is indeed an interaction with U8. Although it is not stated explicitely, 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[6]:

$$\forall\, device : \{outdoor_light_sensor, motion_detector\} \bullet$$
$$fail(device)\, followed_by\, \{inform_user(device), inform_facility_manager(device)\}$$

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.

Requirement NF1. If any outdoor light sensor does not work correctly, the control system for rooms should behave as if the outdoor light sensor had been submitting the last correct measurement of the outdoor light constantly.

Formalization :

$$\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet failed(tr(i).s, outdoor_light_sensor)$$
$$\Rightarrow outdoor_sensor_value(tr(i).s) = last_outdoor_sensor_value(tr(i).s)$$

We have introduced two new functions and a new predicate on the system state. How these are exactly defined must be stated in the formal specification.

Schematic expression : $failed \Rightarrow last_outdoor_sensor_value$

Semantic relations:
$fail \rightsquigarrow failed$
$failed \Rightarrow last_outdoor_sensor_value$
Note that there are no events that falsify the predicate *failed*.

Interaction candidates: U10.
We judge that there is no interaction.

Discussion :
We have detected missing requirements here : The state *failed* of a device should be reversable, i.e., there should be a possibility to repair the system after a failure. Nothing is said about repair in the problem description.

Moreover, we do not see why NF1–NF5 are called "non-functional". In our opinion they are perfectly functional, only that they concern the function of the system in the presence of failures. An example for a requirement that we would call non-functional is NF8.

Requirement NF2. If any outdoor light sensor does not work correctly, the default light scene for all rooms is that all ceiling light groups are on.

[6]For the definition of *followed_by*, see Appendix A.

Formalization :

$$\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet failed(tr(i).s, outdoor_light_sensor) \\ \Rightarrow default_light_scene(tr(i).s) = ceiling_lights_on$$

where $ceiling_lights_on$ is a constant of type $LIGHT_SCENE$.

Schematic expression : $failed \Rightarrow ceiling_light_default$

Semantic relations: $failed \Rightarrow ceiling_light_default$.

Interaction candidates: U10, NF1.
We judge that there is no interaction.

Discussion :
The question arises if a sensor failure permanently overrides the default light scene of a room. In this case, it would have to be freshy defined after a repair, which would not be user-friendly.

Requirement **NF4** is treated similarly.

This concludes the formalization of the requirements and the interaction analysis. In the following, we summarize our results.

4.3 A Summary of Produced Documents

The result of our work consists of several documents:

1. an entity-relationship 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 the Step 3);
3. a list of system operations, concerning the *transformational* part of the system (see results of the step 4);
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 an office: $occupied(room)$, $safe_illumination$, $conforms(light_state, light_scene)$, $failed(device)$.

   The functions defined on the system state are:

   - $light_scene : STATE \rightarrow LIGHT_SCENE$
   - $default_light_scene : STATE \rightarrow LIGHT_SCENE$
   - $light_state : STATE \rightarrow LIGHT_STATE$
   - $outdoor_sensor_value : STATE \rightarrow OUTDOOR_SENSOR_VALUE$
   - $last_outdoor_sensor_value : STATE \rightarrow OUTDOOR_SENSOR_VALUE$

6. tables of semantic relations between event and predicate symbols, see Appendix B.3;
7. a formalization of the requirements, expressed as constraints on admissible system event traces (see Appendix B.1);
8. schematic expressions for the requirements, together with interaction candidates, see Appendix B.2.

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

Performing this case study served us to further validate our approach. In the past, 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].

In the Section 5.1, we discuss the results of our work specific to the case study. In Section 5.2, we discuss the characteristics of our method in general.

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, glossary.

- We could not find out the difference between the *current* and the *chosen* light scene. This difference should be explained in the glossary.
- The difference between between a *light scene* and a *light setting* is not clear for us. In the glossary, 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.
- The definition of a default light scene as a light scene for an unoccupied room seems to contradict U4, which requires the default light scene to be established when be room is *re*occupied.

Unused details. In paragraph 7 of the problem description, *door closed contacts* are mentioned. However, the states or values 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 perchased, 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 an extra section.

Confusion between functional and non-functional requirements. As already noted, we consider NF1–NF5 to be functional. It is not clear to us why they are called non-functional.

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.

Incoherent requirements. Our heuristic 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 additionally 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 completely and unambiguously stated. Our method helps to detect missing, incoherent, and unsatisfiable requirements, as was demonstrated in Section 4. Formalizing the requirements leads to eliminating ambiguities.

Furthermore, formal requirements have a second 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 as small as possible. 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 our goals stated in Section 2). 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. But it is nevertheless important to define separate steps, because they structure the requirements elicitation process and help focus attention to crucial points.

In summary, in our approach we give 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

[AP98] A. I. Antón and C. Potts. The use of Goals to Surface Requirements for Evolving Systems. In *Proc. of the 20 th International Conference on Software Engineering, ICSE'98*, Kyoto, Japan, 1998. I.E.E.E.

[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.v. 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.

[DLF93] A. Dardenne, A.v. Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-Oriented Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[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.

[FKV94] M.D. Fraser, K. Kumar, and V.K. Vaisnavi. Strategies for Incorporating Formal Specifications in Software Development. *CACM*, 37(10):74–86, Oct. 1994.

[FS97] M. Fowler and K. Scott. *UML destilled. Applying the standard object modelling Language*. Addison-Wesley, 1997.

[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 JAcky. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, *Proceedings of the 18t International conference on Conceptual Modelling, ER'99*, number 1728 in LNCS, pages 309–324. Springer-Verlag, 1999.

[HS99b] M. Heisel and J. Souquières. De l'élicitation des besoins à la spécification formelle. *TSI*, 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.

[RBP+91] J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[SH00] Jeanine Souquières and Maritta Heisel. Une méthode pour l'élicitation des besoins : application au système de contrôle d'accès. In Yves Ledru, editor, *Proceedings 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 formal specification of traces, we use the Z notation [Spi92].

$[STATE, EVENT, TIME]$

TraceItem

$s : STATE$
$e : EVENT$
$t : TIME$

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 valid system trace, we require that events later in the sequence do not happen at an earlier time than events earlier in the sequence.

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

Let Tr be the set of admissible traces of a system. For each admissible trace tr, its prefixes are also possible traces :

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

To express constraints concisely, we define several specification macros.

$$
\begin{array}{|l}
_immediately_followed_by_ : EVENT \leftrightarrow EVENT \\
\hline
\forall\, e_1, e_2 : EVENT \bullet \\
\quad e_1\; immediately_followed_by\; e_2 \\
\quad \Leftrightarrow \\
\quad (\forall\, tr : Tr \bullet (\forall\, i : \mathsf{dom}\, tr \mid (tr\, i).e = e_1 \bullet \\
\qquad i = \#tr \lor (tr(i+1)).e = e_2))
\end{array}
$$

We may also want to express that event e entails a set of events that can occur in any order:

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

B Results of the Formalization Process

B.1 Formal Expressions of Requirements

U1: $\forall\, tr : Tr;\; room : ROOM;\; \forall\, i : dom\, tr \bullet occupied(tr(i).s, room) \Rightarrow safe_illumination(tr(i).s, room)$

U2: $\forall\, tr : Tr \bullet occupied(tr(i).s) \Rightarrow tr(i).e \neq turn_off$

U3: $\forall\, tr : Tr \bullet \forall\, i, j : \mathsf{dom}\, tr \mid j > i \land j + 1 \in \mathsf{dom}\, tr \bullet$
$tr(i).e = leave_last \land tr(j).e = enter_first$
$\land (\forall\, k : i \,..\, j \bullet \neg\, occupied(tr(k).s)) \land tr(j).t - tr(i).t \leq T1$
$\Rightarrow light_scene(tr(j+1).s) = light_scene(tr(i).s)$

U4: $\forall\, tr : Tr \bullet \forall\, i, j : \mathsf{dom}\, tr \mid j > i \land j + 1 \in \mathsf{dom}\, tr \bullet$
$tr(i).e = leave_last \land tr(j).e = enter_first$
$\land (\forall\, k : i \,..\, j \bullet \neg\, occupied(tr(k).s)) \land tr(j).t - tr(i).t \geq T1$
$\Rightarrow light_scene(tr(j+1).s) = default_light_scene(tr(i).s)$

U5: $\forall\, tr : Tr \bullet ls : LIGHT_SCENE \bullet \forall\, i : \mathsf{dom}\, tr \mid i < \#tr \bullet$
$tr(i).e = select_light_scene(ls) \Rightarrow light_scene(tr(i+1).s) = ls$

U8: $\forall\, device : \{outdoor_light_sensor, motion_detector\} \bullet$
$fail(device)\, followed_by\, \{inform_user(device), inform_facility_manager(device)\}$

U10: $\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet conforms(light_state(tr(i).s), light_scene(tr(i).s))$
FM3: $\forall\, tr : Tr \bullet \forall\, i, j : \mathsf{dom}\, tr \mid tr(j).t - tr(i).t \geq T3 \bullet$
$tr(i).e = leave_last \wedge (\forall\, k : i+1 \mathinner{..} j \bullet \neg\, occupied(tr(k).s))$
$\Rightarrow (\exists\, l : i+1 \mathinner{..} j \bullet tr(l).e = turn_off \wedge tr(l).t = tr(i).t + T3)$

FM6: $\forall\, tr : Tr \mid \neg\, occupied(tr(\#tr).s) \bullet$
$\exists\, tr' : Tr \bullet front\, tr' = front\, tr \wedge tr'(\#tr).s = tr(\#tr).s \wedge tr'(\#tr).e = turn_off_manual$

FM7: $\forall\, d : DEVICE \bullet fail(d)\, immediately_followed_by\, inform_facility_manager(d)$

NF1: $\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet failed(tr(i).s, outdoor_light_sensor)$
$\Rightarrow outdoor_sensor_value(tr(i).s) = last_outdoor_sensor_value(tr(i).s)$

NF2: $\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet failed(tr(i).s, outdoor_light_sensor)$
$\Rightarrow default_light_scene(tr(i).s) = ceiling_lights_on$

NF4: $\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet failed(tr(i).s, motion_detector) \Rightarrow occupied(tr(i).s)$

B.2 Schematic Constraints and Interaction Candidates

Req.	schematic expressions	Candidates	Interactions
U1	$occupied \Rightarrow safe_illumination$	–	–
U2	$occupied \rightsquigarrow \neg\, turn_off$	U1	–
U3	$leave_last \wedge enter_first \wedge \neg\, occupied \wedge lesseqT1 \rightsquigarrow same_light_scene$	U1, U2	–
U4	$leave_last \wedge enter_first \wedge \neg\, occupied \wedge greaterT1 \rightsquigarrow default_light_scene$	U1, U2	–
U5	$select_light_scene \rightsquigarrow selected_light_scene_established$	U1, U2, U3, U4	U1?
U8	$fail \rightsquigarrow inform_user \wedge$ **inform_facility_manager**	–	–
U10	$true \Rightarrow conforms$	U1–U5, U8	U1?
FM3	$leave_last \wedge \neg\, occupied \rightsquigarrow turn_off$	U1–U5, U10	–
FM6	$\neg\, occupied \rightsquigarrow turn_off_manual$	U3, U4, U10, FM3	FM3?
FM7	$fail \rightsquigarrow inform_facility_manager$	U8, U10	U8
NF1	$failed \Rightarrow last_outdoor_sensor_value$	U10	–
NF2	$failed \Rightarrow ceiling_light_default$	U10, NF1	–
NF4	$failed \Rightarrow occupied$	U10, NF1, NF2	–

B.3 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

$$
\begin{aligned}
\mathit{enter_first} &\leadsto \mathit{occupied}\\
\mathit{leave_last} &\leadsto \neg\ \mathit{occupied}\\
\mathit{enter_first} &\leadsto \mathit{safe_illumination}\\
\mathit{select_light_scene} &\leadsto \mathit{safe_illumination}\\
\mathit{turn_off} &\leadsto \neg\ \mathit{safe_illumination}\\
\mathit{turn_off_manual} &\leadsto \neg\ \mathit{safe_illumination}\\
\mathit{enter_first} &\leadsto \mathit{same_light_scene}\\
\mathit{select_light_scene} &\leadsto \neg\ \mathit{same_light_scene}\\
\mathit{enter_first} &\leadsto \mathit{default_light_scene}\\
\mathit{select_light_scene} &\leadsto \neg\ \mathit{default_light_scene}\\
\mathit{select_light_scene} &\leadsto \mathit{selected_light_scene_established}\\
\mathit{turn_off} &\leadsto \neg\ \mathit{selected_light_scene_established}\\
\mathit{fail} &\leadsto \mathit{failed}
\end{aligned}
$$

Relations between predicate literals

$\mathit{occupied}_{\Rightarrow} = \{\mathit{safe_illumination}\}$
$\mathit{failed}_{\Rightarrow} = \{\mathit{occupied}, \mathit{last_outdoor_sensor_value}, \mathit{ceiling_light_default}, \mathit{safe_illumination}\}$