A Heuristic Approach to Detect Feature Interactions in Requirements : Application to the Lift System

Maritta Heisel[1] and Jeanine Souquières[2]

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

Abstract. We present a method to systematically detect feature interactions in requirements, which are expressed as constraints on system event traces. We show its application on the lift system, incorporating new features to a simple lift, concerning the lift overfull and the executive floor with priority. This method is part of a broader approach to requirements elicitation and formal specification.

1 The General Approach

Our work aims at providing methological support for analysts and specfiers of software-based systems. To this end, we have developed an integrated approach to requirements elicitation and formal specification, which is sketched in [HS98]. We do not invent any new languages, but give guidance how to proceed to (i) identify and formally express the requirements concerning the system to be constructed, and (ii) systematically transform these requirements into a formal specification. The difference between requirements and a specification is that requirements refer to the entire system to be realized, whereas a specification refers only to the part of the system to be implemented by software.

Our method begins with an explicit requirements elicitation phase. The result of this first phase is a set of requirements, which are expressed formally as constraints on sequences of events or operations that can happen or be invoked in the context of the system. These constraints form the starting point for the development of the formal specification. The two phases provide feedback to one another: not only is the specification based on the requirements, but the specification phase may also reveal omissions and errors in the requirements. In the present paper, however, we will not describe the specification phase, because the detection of feature interactions is part of the requirements elicitation phase. Expressing requirements formally greatly supports the systematic detection of feature interactions.

We use *agendas* [Hei98] to express our methods. An agenda is a list of steps to be performed when carrying out some task in the context of software engineering. The result of the task will be a document expressed in a certain language. Agendas contain informal descriptions of the steps. With each step, schematic expressions of the language in which the result of the activity is expressed can be associated. The schematic expressions are instantiated when the step is performed. The steps listed in an agenda may depend on each other. Usually, they will have to be repeated to achieve the goal, because later steps will reveal errors and omissions in earlier steps.

Agendas are not only a means to guide software development activities. They also support quality assurance because the steps of an agenda may have validation

conditions associated with them. These validation conditions state necessary semantic conditions that the artifact must fulfill in order to serve its purpose properly.

Section 2 presents a brief overview of our requirements elicitation method. Section 3 details the incorporation of a single constraint into a set of existing constraints. Section 4 describes the algorithm for detecting interaction candidates. Section 5 illustrates the application of the approach to the integration of new features to a simple lift system. Section 6 presents a discussion of the approach and its benefits.

2 Agenda for Requirements Elicitation

Requirements elicitation is performed in five steps, which provide methodological guidance for analysts. In the following, we list the steps of the agenda we have developed for requirements elicitation. Only the most important validation conditions are mentioned. For more detail on the approach, see [HS99a].

1. Introduce the domain vocabulary. The different notions of the application domain are expressed in a textual form.
2. State the facts, assumptions, and requirements concerning the system in natural language, as a set of fragments corresponding to parts of scenarios of the 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 things 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. These conditions are expressed as *assumptions*.
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] in which events are : (i) controlled by the environment and not shared with the software system, (ii) controlled by the environment but observable by the software system, (iii) controlled by the software system and observable by the environment, and (iv) controlled by the software system and not shared with the environment.
   Validation condition : there must not be any events controlled by the software system and not shared with the environment.
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.
5. Formalize the facts, assumptions, and requirements as constraints on the possible traces of system events.

Using constraints to talk about the behavior of the system has the following advantages:

- It is possible to express *negative* requirements, i.e., to require that certain things do not happen. Such constraints are often related to safety conditions of the system to be realized.

- It is possible to give scenarios, i.e., example behaviors of the system [FS97]. Such constraints are often related to liveness conditions for the system to be realized.
- Giving constraints do not fix the system behavior entirely. They do not restrict the specification unnecessarily. Any specification that fulfills the constraints is admitted [HS99b].

Adding constraints may either restrict or enlarge the set of possible system behaviors.

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.

3 Agenda to Incorporate Single Constraints

In Step 5 of the agenda for requirements elicitation, 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 behaviors [HS98]. The following agenda gives guidelines how to incorporate a new constraint into a set of already existing constraints.

Our method is a heuristic one, in the sense it does not exist a rigourous definition of the interaction concept. Unwanted behaviours are the choice of the customer.

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*. *Event literals* are defined analogously.

1. Formalize the new constraint as a formula on system 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 e occurs. For each possible trace, its prefixes are also possible traces. A formal specification of traces is defined 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.
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 have an invariant of the system.

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 Section 4. Interaction analysis and its automation needs to manipulate schematic expressions of formalized constraints. These schematic expressions have the following 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 $\diamond$ denotes either conjunction or disjunction. The $\rightsquigarrow$ symbol 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.

3. Update the tables of semantic relations. The detection of constaint interactions cannot be based on syntax alone, the algorithm being completely automatic, without anay user intervention. 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 event establish and which events falsify it. We construct three tables of semantic relations:
   (a) Necessary conditions for events. If an event e can only occur if predicate literal pl is true, then this table has an entry $pl \leftsquigarrow e$.
   (b) Events establishing predicates. For each predicate literal pl, we need to know the events e that establish it: $e \rightsquigarrow pl$
   (c) 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: $\neg\ p_{\Rightarrow} = \{q : PLit \mid \neg\ p \Rightarrow q\}$

      These tables are not only useful to detect interactions; they are also useful to develop and validate the formal specification of the software system.
4. Determine interaction candidates, based on the list of schematic requirements (Step 2) and the semantic relation tables (Step 3). The definition of the interaction candidates is given in Section 4.
5. Decide if there are interactions of the new constraint with the determined candidates. The algorithm determines a set of candidates to examine. It does not proof that an interaction exists between the new constraint and ech 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 behaviour or not.

6. Resolve interactions. To resolve an interaction, it is usual to relax requirements or to 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 of the agenda for requirements elicitation :

- each requirements of Step 2 must be expressed,
- the set of constraints must be consistent,
- for each introduced predicate, events that modify it must be observable by the software system.

Step 1 to step 6 preserve the mutual coherence between the set of constraints. Usually, revisions and communication with customers will be necessary.

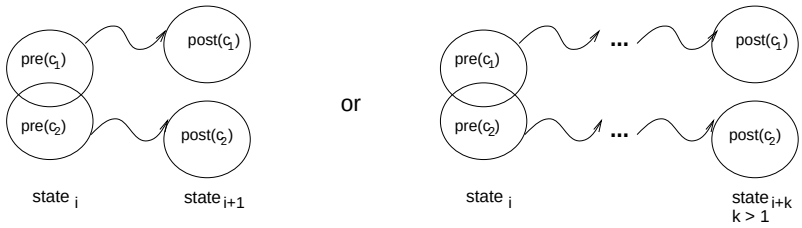

Fig. 1. Interaction candidates

4 Determining Interaction Candidates

Our method to determine interaction candidates is based on the following observations : two constraints are interaction candidates for one another if they have common preconditions but incompatible postconditions, as is illustrated in Figure 1.

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.

Constraints[1] $\underline{x} \rightsquigarrow \underline{y}$ and $\underline{u} \rightsquigarrow \underline{w}$ are possible interaction candidates, when their preconditions ($\underline{x}$ and $\underline{u}$) are neither exclusive nor independent of each other. This

[1] Underlined identifiers denote sets of literals.

means, there are situations where both $\underline{x} \rightsquigarrow \underline{y}$ and $\underline{u} \rightsquigarrow \underline{w}$ might apply. If in such a case the postconditions ($\underline{y}$ and $\underline{w}$) are incompatible, we have found an interaction.

Constraints $\underline{x} \rightsquigarrow \underline{y}$ and $\underline{u} \rightsquigarrow \underline{w}$ may interact on the postcondition, if we can find a literal l such that $\underline{y}$ entails l and $\underline{w}$ entails $\neg\, l$. If in such a case the preconditions $\underline{x}$ and $\underline{u}$ do not exclude each other, an interaction occurs.

4.1 Precondition Interaction

Two constraints $\underline{x} \rightsquigarrow \underline{y}$ and $\underline{u} \rightsquigarrow \underline{w}$ have common literals in their precondition ($\underline{x} \cap \underline{w} \neq \varnothing$), 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 p, and e is only possible if p holds ($p \leftsquigarrow e$), then we also have detected a common precondition between the two events.

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 there is a predicate literal w with $p \Rightarrow w$ and $q \Rightarrow w$, then w is a common precondition.

Figure 2 shows the general approach to find interaction candidates $C_{pre}(c', far)$ by a precondition analysis for a new constraint c' among the *far* facts, assumptions, and requirements already defined.

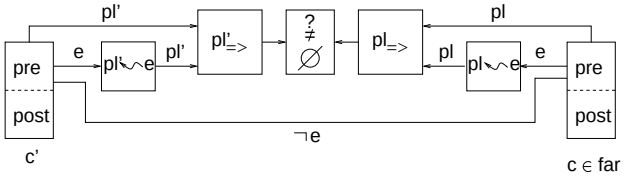

Fig. 2. Determining interaction candidates by precondition analysis

Let be

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

True predicate 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 symbol $e \in precond(c)$:

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

The complete precondition of a constraint c results from the transitive and reflexive closure of the $pre\_predicates(c)$ set with respect to the implication, i.e.

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

A constraint $c \in far$ is an interaction candidate with a new constraint c' if their preconditions or their complete preconditions only contain common literals.

$$C_{pre}(c', far) =$$
$$\{c : far \mid precond(c) \cap precond(c') \neq \varnothing\}$$
$$\cup$$
$$\{c : far \mid \exists pl : pre\_predicates(c);\ pl' : pre\_predicates(c') \bullet pl_{\Rightarrow} \cap pl'_{\Rightarrow} \neq \varnothing\}$$

Two cases are distinguished because the precondition of a constraint can contain event literals, whereas the complete 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, and that the candidate function distributes over set union of the preconditions of contraints:

$$\forall c, c_1, c_2 : Constraint;\ cs : \mathbb{P}\, Constraint \bullet$$
$$c_2 \in C_{pre}(c_1, cs \cup \{c_2\}) \Leftrightarrow c_1 \in C_{pre}(c_2, cs \cup \{c_1\})$$
$$\wedge$$
$$precond(c) = precond(c_1) \cup precond(c_2) \Rightarrow$$
$$C_{pre}(c, cs) = C_{pre}(c_1, cs) \cup C_{pre}(c_2, cs)$$

The latter implies that, when a constraint is changed by adding a new literal to its precondition, the interaction analysis has to be performed only on this new precondition.

4.2 Postcondition Interaction

To find conflicting postconditions, we perform the complete postcondition of the new constraint c' and the one of each constraint $c \in far$ in the same way as for the preconditions. A constraint c is an interaction candidate with the new constraint c' if there exists a literal x in its postcondition or in its complete postcondition, the negation of which is in the postcondition or in the complete postcondition of c'. Figure 3 gives an overview of the procedure.

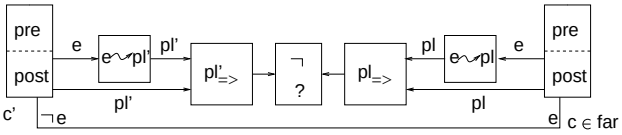

Fig. 3. Determining interaction candidates by postcondition analysis

We need the auxiliary definitions

$$postcond(x_1 \diamond x_2 \diamond \ldots \diamond x_n \rightsquigarrow y_1 \diamond y_2 \diamond \ldots \diamond y_k) = \{y_1, \ldots y_k\}$$
$$e_{\rightsquigarrow} = \{pl : PLit \mid e \rightsquigarrow pl\}$$
$$post\_predicates(c) = (postcond(c) \cap PLit) \cup \bigcup_{e \in postcond(c) \cap EVENT} e_{\rightsquigarrow}$$
$$ls_1\ opposite\ ls_2 \Leftrightarrow \exists x : ls_1 \bullet \neg x \in ls_2$$

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

Now, we can define

$$
\begin{array}{l}
C_{post}(c', far) = \\
\quad \{c : far \mid postcond(c)\ opposite\ postcond(c')\} \\
\quad \cup \\
\quad \{c : far \mid \exists\, pl : post\_predicates(c);\ pl' : post\_predicates(c') \bullet pl_{\Rightarrow}\ opposite\ pl'_{\Rightarrow}\}
\end{array}
$$

Of course, this definition is symmetric, too, and C_{post} distributes over set union of postconditions of contraints.

5 Example: the Lift System

We first consider a simple lift with the following requirements:

1. The lift is called by pressing a button.
2. Pressing a call button is possible any time.
3. A call is served when the lift arrives at the corresponding floor.
4. When the lift passes by a floor f, and there is a call from this floor, then the lift will stop at this floor.
5. When the lift has stopped, it will open the door.
6. When the lift door has been opened, it will close automatically after d time units.
7. The lift only changes its direction when there are no more calls in the current direction.
8. When the lift is halted at a floor with the door open, a call for this floor is not taken into account.
9. When the lift is halted at a floor with the door closed and receives a call for this floor, it opens its door.
10. Whenever the lift moves, its door must be closed.

As a fact, we formalize that the door can only be opened when it is closed and vice versa. Afterwards, we will add the following features:

11. When the lift is overloaded, the door will not close. Some passengers must get out.
12. The lift gives priority to calls from the executive landing.

5.1 Starting Point

The following tables present the schematic constraints for the fact and Requirements 1–10, and the corresponding tables of semantic relations. The formalized fact and Requirements 1–10 are given in Appendix B.

The schematic constraints, see Step 2 of the agenda of Section 3, are given in Table 1. In the formal expressions corresponding to *fact*, req_1 and req_7, the precondition refers to a later state than the postcondition, because necessary conditions for events to happen or predicates to be true are expressed. Our algorithm for feature interaction detections, however, requires the precondition to refer to an earlier or

the same state as the postcondition. Hence, the schematic expression for *fact*, req_1 and req_7 are based on the contraposition of the constraints given in Appendix B (i.e. $\neg\, Q \Rightarrow \neg\, P$ instead of $P \Rightarrow Q$).

Table 2 shows the necessary conditions for the events. The events establishing the predicates and their negations are given in Table 3. Finally, Table 4 gives the implicative closures of the various predicate literals. This information is collected when performing Step 3 of the agenda of Section 3.

Constraint	schematic expression
fact	$\neg$ *door_closed* $\leadsto \neg$ *open* $\neg$ *door_open* $\leadsto \neg$ *close*
req_1	$\neg$ *press* $\leadsto \neg$ *call*
req_2	*true* $\leadsto$ *press*
req_3	*at* $\leadsto \neg$ *call*
req_4	*passes_by* $\wedge$ *call* $\leadsto$ *stop*
req_5	*stop* $\leadsto$ *open*
req_6	*open* $\leadsto$ *close*
req_7	*direction_up* $\wedge$ *call_from_up* $\leadsto$ *direction_up* *direction_down* $\wedge$ *call_from_down* $\leadsto$ *direction_down*
req_8	*halted* $\wedge$ *at* $\wedge$ *door_open* $\wedge$ *press* $\leadsto \neg$ *call*
req_9	*halted* $\wedge$ *at* $\wedge$ *door_closed* $\wedge$ *press* $\leadsto$ *open*
req_{10}	$\neg$ *halted* $\leadsto$ *door_closed*

Table 1. Overview of schematic constraints

door_open	$\leadsto$ *close*	*halted*	$\leadsto$ *move*
at	$\leadsto$ *move*	*door_closed*	$\leadsto$ *open*
call	$\leadsto$ *move*	$\neg$ *halted*	$\leadsto$ *stop*
door_closed	$\leadsto$ *move*	*passes_by*	$\leadsto$ *stop*

Table 2. Necessary conditions for events

5.2 Adding new features

We now incorporate the features of overloading and executive floor, following the agenda of Section 3.

Requirement 11:

When the lift is overloaded, the door will not close. Some passengers must get out.

$stop \leadsto at$	$close \leadsto door\_closed$
$move \leadsto \neg\ at$	$open \leadsto \neg\ door\_closed$
$press \leadsto call$	$open \leadsto door\_open$
$stop \leadsto \neg\ call$	$close \leadsto \neg\ door\_open$
$press \leadsto call\_from\_up$	$stop \leadsto halted$
$stop \leadsto \neg\ call\_from\_up$	$move \leadsto \neg\ halted$
$press \leadsto call\_from\_down$	$move \leadsto passes\_by$
$stop \leadsto \neg\ call\_from\_down$	$stop \leadsto \neg\ passes\_by$

Table 3. Events establishing predicate literals

$$
\begin{aligned}
at_{\Rightarrow} &= \{halted, \neg\ passes\_by, \neg\ call\}\\
\neg\ at_{\Rightarrow} &= \{passes\_by, \neg\ halted, door\_closed, \neg\ door\_open\}\\
call_{\Rightarrow} &= \varnothing\\
\neg\ call_{\Rightarrow} &= \{\neg\ call\_from\_up, \neg\ call\_from\_down\}\\
call\_from\_down_{\Rightarrow} &= \{call\}\\
\neg\ call\_from\_down_{\Rightarrow} &= \varnothing\\
call\_from\_up_{\Rightarrow} &= \{call\}\\
\neg\ call\_from\_up_{\Rightarrow} &= \varnothing\\
door\_closed_{\Rightarrow} &= \{\neg\ door\_open\}\\
\neg\ door\_closed_{\Rightarrow} &= \{door\_open, halted, at, \neg\ passes\_by\}\\
door\_open_{\Rightarrow} &= \{\neg\ door\_closed, halted, at, \neg\ passes\_by\}\\
\neg\ door\_open_{\Rightarrow} &= \{door\_closed\}\\
halted_{\Rightarrow} &= \{at, \neg\ passes\_by\}\\
\neg\ halted_{\Rightarrow} &= \{passes\_by, \neg\ at, door\_closed, \neg\ door\_open\}\\
passes\_by_{\Rightarrow} &= \{\neg\ at, \neg\ halted, door\_closed, \neg\ door\_open\}\\
\neg\ passes\_by_{\Rightarrow} &= \{at, halted\}
\end{aligned}
$$

Table 4. Relations between predicate literals

Step 1: Formalize the new constraint as a formula on system traces.
$\forall\, tr : Tr \bullet (\forall\, i : \mathsf{dom}\, tr \bullet overloaded(tr(i).s) \Rightarrow door\_open(tr(i).s))$

Step 2: Give a schematic expression of the constraint. $overloaded \leadsto door\_open$

Step 3: Update the tables of semantic relations. With this constraint, we have introduced a new predicate symbol *overloaded* for which we must specify the events that modify it. Hence, we must introduce two new events *enter* and *leave*. We add the lines

$enter \leadsto overloaded$ $\qquad$ $leave \leadsto \neg\ overloaded$

to Table 3 and the lines

$door\_open \leftsquigarrow enter$ $\qquad$ $door\_open \leftsquigarrow leave$

to Table 2. Table 4 must be changed in the following way : we add the lines

$$
\begin{aligned}
overloaded_{\Rightarrow} &= \{door\_open, \neg\ door\_closed, halted, at, \neg\ passes\_by\}\\
\neg\ overloaded_{\Rightarrow} &= \varnothing
\end{aligned}
$$

The entries of all predicates related to *overloaded* must be updated. We get the following changes:

$$\begin{aligned}
\neg\ door_open_{\Rightarrow} &= \{door_closed, \neg\ \mathbf{overloaded}\} \\
door_closed_{\Rightarrow} &= \{\neg\ door_open, \neg\ \mathbf{overloaded}\} \\
\neg\ halted_{\Rightarrow} &= \{passes_by, \neg\ at, door_closed, \neg\ door_open, \neg\ \mathbf{overloaded}\} \\
\neg\ at_{\Rightarrow} &= \{passes_by, \neg\ halted, door_closed, \neg\ door_open, \neg\ \mathbf{overloaded}\} \\
passes_by_{\Rightarrow} &= \{\neg\ at, \neg\ halted, door_closed, \neg\ door_open, \neg\ \mathbf{overloaded}\}
\end{aligned}$$

Step 4: Determine interaction candidates. To determine the precondition interaction candidates, we determine the sets used in the definition of C_{pre} in Section 4.1:

$$pre\_predicates(req_{11}) = \{overloaded\}$$

Hence, the precondition interaction candidates are the ones that have one of the elements $door\_open, \neg\ door\_closed, halted, at, \neg\ passes\_by$ in their precondition. According to Table 1, these are *fact* because of $\neg\ door\_closed$, req_2 because of *true*, req_8 and req_9 because of *halted*.

To determine the postcondition interaction candidates, we proceed according to the definition of C_{post} in Section 4.2:

$$post\_predicates(req_{11}) = \{\neg\ door\_open\}$$

Because $door\_open_{\Rightarrow} = \{\neg\ door\_closed, halted, at, \neg\ passes\_by\}$, we must look for postconditions $door\_closed, passes\_by, \neg\ halted, \neg\ at$ and related events according to Table 3. These are *close* and *move*. According to Table 1, we get the candidates and req_6 req_{10}.

Step 5: Analyze possible interactions. We do not have interactions with *fact*, req_8, req_9, req_{10}, but with req_6, because the door will not close automatically after d units time if the lift is overloaded.

Step 6: Eliminate interactions, if necessary. The new definition of req_6 is:

$$\begin{aligned}
\forall\, tr : Tr \bullet\ & \forall\, i : \mathsf{dom}\, tr \bullet\ tr(i).e = open \wedge last(tr).t > tr(i).t + d \\
& \Rightarrow \exists\, j : \mathsf{dom}(tr(j).t \leq tr(i).t + d \wedge tr(j+1).t > tr(i).t + d \\
& \qquad \vee\ \neg\ \mathbf{overloaded}(\mathbf{tr}(\mathbf{j}).\mathbf{s})) \\
& \qquad\quad \Rightarrow tr(j).e = close \wedge tr(j).t = tr(i).t + d)
\end{aligned}$$

Informal requirements req_6 has to be updated: when the lift door has been opened, it will close automatically after d time units if the lift is not overloaded.

The new schematic constraint becomes

$$open \rightsquigarrow closed \vee overloaded$$

Since we have added the new postcondition *overloaded* to the constraint, we must now perform postcondition interaction analysis on this literal. With $overloaded_{\Rightarrow} =$

$\{door\_open, \neg\ door\_closed, halted, at, \neg\ passes\_by\}$ it follows that we must look for constraints with postconditions $\neg\ door\_open, door\_closed, \neg\ halted, \neg\ at, passes\_by$. Related events according to Table 3 are *close* and *move*. In Table 1, we find the candidates req_{10}. There is no interaction with it.

Requirement 12:

The lift gives priority to calls from the executive landing.

Step 1: Formalize the new constraint as a formula on system traces.
$\forall\, tr : Tr \bullet (\forall\, i : \mathsf{dom}\, tr \bullet call(tr(i).s, executive\_floor)$
$\Rightarrow next\_stop(tr(i).s) = executive\_floor)$

Step 2: Give a schematic expression of the constraint.
$call \Rightarrow next\_stop\_at\_executive\_floor$

Step 3: Update the tables of semantic relations. With this constraint, we introduce a new predicate symbol *next_stop_at_executive_floor* for which we must specify the events that modify it. We add the lines

$$press \rightsquigarrow next\_stop\_at\_executive\_floor \quad stop \rightsquigarrow \neg\ next\_stop\_at\_executive\_floor$$

to Table 3. We get the following entry Table 4 :

$$next\_stop\_at\_executive\_floor_{\Rightarrow} = \{call\}$$

Step 4: Determine interaction candidates. To determine the precondition interaction candidates, we determine the sets used in the definition of C_{pre} in Section 4.1:

$$pre\_predicates(req_{12}) = \{call\}$$

Hence, the precondition interaction candidates are the ones that have one of the elements $call, call\_from\_up, call\_from\_down$ in their precondition. According to Table 1, these are req_4 and req_7.

The postcondition interaction candidates are the ones who have $\neg\ call$ in the postcondition. According to Table 1 we get req_1, req_3 and req_8.

Step 5: Analyze possible interactions. We have interactions with req_4 and req_7, but not with req_3 and req_8, because req_{12} gives priority to the executive floor and not to the current floor as expressed in req_4 or to the current direction as expressend in req_7.

Step 6: Eliminate interactions, if necessary. To adjust req_4, we add a new precondition to it; req_4 becomes

$$\begin{array}{l}\forall\, tr : Tr \bullet (\mathbf{let}\, tr' == remove(tr, \{b : Button \bullet press(b)\}) \bullet \\ \quad \forall\, i : \mathsf{dom}\, tr';\ \mathbf{k} : \mathbf{FLOOR} \mid i \neq \#tr' \bullet \\ \qquad passes\_by(tr'(i).s, \mathbf{k}) \wedge call(tr'(i).s, \mathbf{k}) \\ \qquad \wedge\ (\mathbf{k} = \mathbf{executive\_floor} \vee \neg\ \mathbf{call}(\mathbf{tr'(i).s}, \mathbf{executive\_floor})) \\ \qquad\quad \Rightarrow tr'(i+1).e = stop(\mathbf{k}))\end{array}$$

The new schematic expression for req_4 is:
passes_by $\wedge$ *call* $\wedge$ *next_executive_floor* $\rightsquigarrow$ *stop*
passes_by $\wedge$ *call* $\wedge$ $\neg$ *call* $\rightsquigarrow$ *stop*

Note that now we have *call* as well as $\neg$ *call* in the schematic precondition of the constraint. This is not a contradition (*call* and $\neg$ *call* have different arguments), but only enlarges the set of possible interaction candidates.

We must now perform a precondition interaction analysis on the new precondition $\neg$ *call*. Because there are no related events, our candidates are the constraints with precondition $\neg$ *call*, $\neg$ *call_from_up*, $\neg$ *call_from_down*. There are no interaction candidates.

To adjust req_7, we also add new preconditions.

$$\begin{array}{l}\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \mid i \neq \#tr \bullet \\ \quad (direction(tr(i).s) = up \wedge direction(tr(i+1).s) = down \\ \qquad \Rightarrow (\neg\ call\_from\_up(tr(i).s) \vee\ \mathbf{call}(\mathbf{tr(i).s}, \mathbf{executive\_floor}))) \\ \quad \wedge \\ \quad (direction(tr(i).s) = down \wedge direction(tr(i+1).s) = up \\ \qquad \Rightarrow (\neg\ call\_from\_down(tr(i).s) \vee\ \mathbf{call}(\mathbf{tr(i).s}, \mathbf{executive\_floor})))\end{array}$$

The new schematic expressions for req_7 are
direction_up $\wedge$ *call_from_up* $\wedge$ $\neg$ *call* $\rightsquigarrow$ *direction_up*
direction_down $\wedge$ *call_from_down* $\wedge$ $\neg$ *call* $\rightsquigarrow$ *direction_down*

As for req_4, we must perform a precondition interaction analysis on the new precondition $\neg$ *call*. This yields the same candidates as before, plus the new version of req_4. Again, there is no further interaction.

6 Discussion

The approach for the detection of feature interactions we have presented is truly heuristic. This means, we cannot guarantee that all interactions that might occur are found by our procedure. The virtue of our approach lies in the fact that interactions on the requirements level can be detected very early, before the formal specification is set up, and with relatively little effort. Even though determining the interaction candidates is tedious if performed by hand, the procedures to determine the sets C_{pre} and C_{post} as defined in Section 4 are very easy to implement. Theorem proving

techniques are unnecessary. The number of interaction candidates that are yielded by our procedure and that must be inspected is much less than if a complete analysis were performed.

The semantic information collected in the tables of necessary conditions for events, events establishing predicate literals, and relations between predicate literals not only contributes to a better understanding of the requirements, but also greatly facilitates the process of setting up and validating a formal specification for the software system to be built.

Our approach to detect feature interactions is independent of the order in which the features are added. We do not attempt to resolve feature interactions automatically. Such decisions are best taken by the customers.

References

[FS97] M. Fowler and K. Scott. *UML distilled. Applying the standard Object Modelling Language*. Addison-Wesley, 1997.

[Hei98] M. Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32, London, 1998. Chapman & Hall.

[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 and M. Bouzeghoub and I. Comyn-Wattiau and E. Métais, editor, *Proceedings of the 18th International Conference on Conceptual Modeling*, LNCS 1728, pages 309–324. Springer Verlag, November 1999.

[HS99b] M. Heisel and J. Souquières. De l'élicitation des besoins à la spécification formelle. *TSI*, 18(7), 1999.

[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.

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

A Formal Expression of Constraints on Traces

In the following formal treatment 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 (reflexivity, transitivity, and anti-symmetry).

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.

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

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.

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

The function *remove* takes a trace and a set of events as its arguments and removes all trace elements whose event is in the given set.

$$\begin{array}{|l} remove : TRACE \times \mathbb{P}\, EVENT \rightarrow TRACE \\ \hline \forall\, tr : TRACE;\ evs : \mathbb{P}\, EVENT \bullet remove(tr, evs) = tr \upharpoonright \{ti : TraceItem \mid ti.e \notin evs\} \end{array}$$

B Formal Versions of Requirements and Facts

Fact

The door can only be opened when it is closed and vice versa.

$$\begin{array}{l} \forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet \\ \qquad (tr(i).e = open \Rightarrow tr(i).s = door\_closed) \wedge \\ \qquad (tr(i).e = close \Rightarrow tr(i).s = door\_open) \end{array}$$

Requirement 1:

The lift is called by pressing a button.

$$\begin{array}{l} \forall\, tr : Tr \bullet (\forall\, i : \mathsf{dom}\, tr;\ b : BUTTON \bullet \\ \quad call(tr(i).s, floor(b)) \Rightarrow (\exists\, j : \mathsf{dom}\, tr \mid j < i \bullet tr(j).e = press(b))) \end{array}$$

Requirement 2:

Pressing a call button is possible any time.

$$\forall\, tr : Tr;\ b : BUTTON \bullet (\exists\, tr' : Tr \bullet front(tr') = tr \wedge last(tr').e = press(b))$$

Requirement 3:

A call is served when the lift arrives at the corresponding floor.

$$\forall\, tr : Tr \bullet (\forall\, i \,\mathsf{dom}\, tr;\ f : FLOOR \bullet at(tr(i).s, f) \Rightarrow \neg\, call(tr(i).s, f))$$

Requirement 4:

When the lift passes by a floor f, and there is a call from this floor, then the lift will stop at this floor.

$$\begin{array}{l}\forall\, tr : TR;\ f : FLOOR \bullet (\mathbf{let}\ tr' == remove(tr, \{b : BUTTON \bullet press(b)\}) \bullet \\ \quad \forall\, i : \mathsf{dom}\, tr' \mid i \neq \#tr' \bullet \\ \quad\quad passes\_by(tr'(i).s, f) \wedge call(tr'(i).s, f) \Rightarrow tr'(i+1).e = stop(f))\end{array}$$

Because *press* events are always possible, we must remove them from the traces when we want to express liveness conditions for the lift.

Requirement 5:

When the lift has stopped, it will open the door.

$$\begin{array}{l}\forall\, tr : Tr;\ f : FLOOR \bullet (\mathbf{let}\ tr' == remove(tr, \{b : BUTTON \bullet press(b)\}) \bullet \\ \quad \forall\, i : \mathsf{dom}\, tr' \mid i \neq \#tr' \bullet tr'(i).e = stop(k) \Rightarrow tr'(i+1).e = open)\end{array}$$

Requirement 6:

When the lift door has been opened, it will close automatically after d time units.

$$\begin{array}{l}\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \bullet tr(i).e = open \wedge last(tr).t > tr(i).t + d \\ \quad\quad \Rightarrow \exists\, j : \mathsf{dom}\, tr \bullet tr(j).e = close \wedge tr(j).t = tr(i).t + d\end{array}$$

Requirement 7:

The lift only changes its direction when there are no more calls in the current direction.

$$\begin{array}{l}\forall\, tr : Tr \bullet \forall\, i : \mathsf{dom}\, tr \mid i \neq \#tr \bullet \\ \quad (direction(tr(i).s) = up \wedge direction(tr(i+1).s) = down \\ \qquad \Rightarrow \neg\ call\_from\_up(tr(i).s)) \\ \quad \wedge \\ \quad (direction(tr(i).s) = down \wedge direction(tr(i+1).s) = up \\ \qquad \Rightarrow \neg\ call\_from\_down(tr(i).s))\end{array}$$

Requirement 8:

When the lift is halted at a floor with the door open, a call for this floor is not taken into account.

$$\begin{array}{l}\forall\, tr : Tr;\ b : BUTTON \bullet \forall\, i : \mathsf{dom}\, tr \mid i \neq \#tr \bullet \\ \quad halted(tr(i).s) \wedge at(tr(i).s, floor(b)) \wedge door\_open(tr(i).s) \wedge tr(i).e = press(b) \\ \qquad \Rightarrow \neg\ call(tr(i+1).s, floor(b))\end{array}$$

Requirement 9:

When the lift is halted at a floor with the door closed and receives a call for this floor, it opens its door.

The following formula expresses that the first event different from *press* is *open*.

$$\begin{array}{l}\forall\, tr : Tr;\ b : BUTTON \bullet \forall\, i \in \mathsf{dom}\, tr \bullet \\ \quad halted(tr(i).s) \wedge at(tr(i).s, floor(b)) \wedge door\_closed(tr(i).s) \wedge tr(i).e = press(b) \\ \qquad \Rightarrow ((\exists\, j : \mathsf{dom}\, tr \bullet j > i \wedge \forall\, b : BUTTON \bullet tr(j) \neq press(b)) \\ \qquad\quad \Rightarrow (\exists\, k : \mathsf{dom}\, tr \mid k > i \bullet tr(k).e = open \wedge \\ \qquad\qquad\qquad \forall\, l \in i+1\,..\,k-1 \bullet \exists\, b : BUTTON \bullet tr(l).e = press(b)))\end{array}$$

Requirement 10:

Whenever the lift moves, its door must be closed.

$$\forall\, tr : Tr \bullet (\forall\, i : \mathsf{dom}\, tr \bullet \neg\ halted(tr(i).s) \Rightarrow door\_closed(tr(i).s))$$
