Detecting Feature Interactions – A Heuristic Approach

Maritta Heisel
Fakultät für Informatik
Universität Magdeburg
D-39016 Magdeburg, Germany
Fax: (49)-391-67-12810
heisel@cs.uni-magdeburg.de

Jeanine Souquières
LORIA—Université Nancy2
B.P. 239 Bâtiment LORIA
F-54506 Vandœuvre-les-Nancy, France
Fax: (33)-3-83-41-30-79
souquier@loria.fr

Abstract

We present a method to systematically detect feature interactions in requirements. The requirements are expressed as constraints on system event traces. This method part is part of a broader approach to requirements elicitation and formal specification.

1 The General Approach

Our work aims at providing methodological support for analysts and specifiers 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 our method to detect 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. These 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.

2 Agenda for Requirements Elicitation

Requirements elicitation is performed in six 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.

1. Introduce the domain theory.
   All necessary notions must be introduced. These can either be entities, corresponding to nouns in a natural-language description, or relationships, corresponding to verbs in a natural-language description.

2. List all possible events that can happen in connection with the system, together with their parameters.

3. Classify the events as: (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 possible system operations that can be invoked by users, together with their input and output parameters. Introduce a relation between the input and output parameters.

5. State the facts, assumptions, and requirements concerning the system in natural language. 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*.

6. 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. Such constraints are often related to liveness conditions for the system to be realized.

- Giving constraints does not fix the system behavior entirely. Constraints do not restrict the specification unnecessarily. Any specification that fulfills them is permitted.

Note that adding constraints may not only restrict but also enlarge the set of possible system behaviors.

3 Agenda to Incorporate Single Constraints

In Step 6 of the agenda for requirements elicitation, the constraints must be formalized one by one. Each new constraint is added to the set of constraints defined so far. But before the constraint is added, its possible interactions with other constraints should be analyzed. The

following agenda gives guidelines how to incorporate a new constraint into a set of already existing constraints.

Our method is a heuristic one, which means that we cannot guarantee that all interactions are detected. Our aim is to provide a simple procedure that works well in practical cases and that may be applied when a complete interaction analysis is unfeasible.

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.
   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).

2. Give a schematic expression of the constraint.
   These schematic expressions have the following form:

   $$x_1 \wedge x_2 \wedge \ldots \wedge x_n \rightsquigarrow y_1 \vee y_2 \vee \ldots \vee y_k$$

   where the x_i, y_j are literals. The symbol $\rightsquigarrow$ indicates that the precondition refers to an earlier state as the postcondition. If the constraint is an invariant of the system state, then the corresponding schema has the form

   $$x_1 \wedge x_2 \wedge \ldots \wedge x_n \Rightarrow y_1 \vee y_2 \vee \ldots \vee y_k$$

   where the x_i, y_j are predicate literals. The use of the implication symbol $\Rightarrow$ indicates that pre-and postcondition refer to the same state.

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

   (a) Necessary conditions for events.
   If an event e can only occur if predicate literal pl is true, then this table has an entry $pl \rightsquigarrow e$.

   (b) Events establishing predicate literals.
   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: $\neg\ p_{\Rightarrow} = \{q : PLit \mid \neg\ p \Rightarrow q\}$
   - 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: ${}_{\Rightarrow\neg}\ p = \{q : PLit \mid q \Rightarrow \neg\ p\}$

   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. To check consistency of the sets, the equivalences

$$p \in q_{\Rightarrow} \;\Leftrightarrow\; q \in {}_{\Rightarrow}p \;\Leftrightarrow\; \neg\, q \in \neg\, p_{\Rightarrow}$$

can be used.

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. It is up to the analysts and the customers to decide if the conjunction of the new with the candidates yield an unwanted behavior or not.

6. If an interaction occurs, take one of the following actions:

   - correct a fact
   - relax a requirement (usually by adding a new pre- or postcondition, as preconditions are usually conjunctions, and postconditions are usually disjunctions)
   - strengthen an assumption

   Perform an interaction analysis on those literals that were changed or newly introduced into the constraint.

4 Determining Interaction Candidates

Our method to determine interaction candidates is based on the following observations: Constraint interactions can manifest themselves in the pre- or in the postcondition of constraints. Constraints $\underline{x} \rightsquigarrow \underline{y}$ 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 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

To decide if two constraints $\underline{x} \rightsquigarrow \underline{y}$ and $\underline{u} \rightsquigarrow \underline{w}$ might interact on their precondition, we perform the following reasoning: if the two constraints have common literals in their precondition ($\underline{x} \cap \underline{w} \neq \emptyset$), 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 ($p \rightsquigarrow e$), 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 $p \Rightarrow q$ or vice versa, then there is a common precondition.

Figure 1 shows the general approach to find interaction candidates of the precondition for a new constraint c' among the facts, assumptions, and requirements already defined.

To formally define the set $C_{pre}(c', far)$ of candidates of precondition interaction of a new constraint c' with respect to a set far of constraints representing facts, assumptions, and requirements, we first introduce some auxiliary definitions: For each event e, predicate literal pl

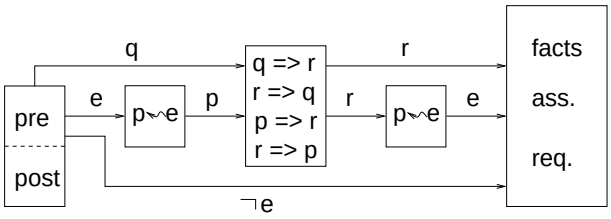

Figure 1: Candidates for precondition interaction

and constraint c, we define

$$\rightsquigarrow e = \{pl : PLit \mid pl \rightsquigarrow e\}$$

$$pre\_predicates(c) = (precond(c) \cap PLit) \cup \bigcup_{e \in precond(c) \cap EVENT} \rightsquigarrow e$$

With these preliminaries, we can define

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

This definition can be explained as follows: All constraints c with a common literal in the precondition are candidates. For events e in the precondition of c', all predicates that are necessary for e to occur are collected. Together with the predicate literals contained in c''s precondition, they form the set $pre\_predicates(c')$. For each $x \in pre\_predicates(c')$, the transitive closure with respect to implication is computed, where both forward ($x_{\Rightarrow}$) and backward chaining ($\Rightarrow x$) are performed. This is necessary because weaker as well as stronger literals have states in common with x. Moreover, this ensures that the candidates are independent of the order in which the constraints are added. Each constraint c whose precondition contains an element of the transitive closure of some x is a candidate. But also those c that contain in their precondition an event e that has a necessary precondition contained in the transitive closure of some x must be added to the set of candidates.

Note that on event literals $\neg\ e$ no chaining is performed, because it is impossible to infer anything from the non-occurrence of an event.

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 constraints:

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

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 literal.

4.2 Postcondition Interaction

For determining the 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 ($e \rightsquigarrow p$) is used. Again, on negative event literals, no chaining is performed. Figure 2 gives an overview of the procedure.

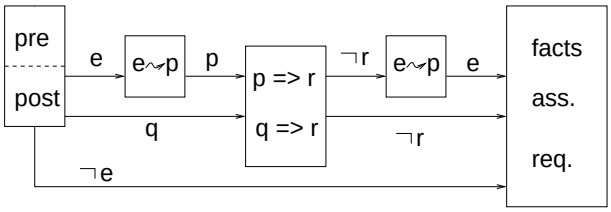

Figure 2: Candidates for precondition interaction

We need the auxiliary definitions

$$e_{\rightsquigarrow} = \{pl : PLit \mid e \rightsquigarrow pl\}$$
$$post\_predicates(c) = (postcond(c) \cap PLit) \cup \bigcup_{e \in postcond(c) \cap EVENT} e_{\rightsquigarrow}$$
$$ls_1 \; opposite \; ls_2 \Leftrightarrow \exists x : ls_1 \bullet \neg x \in ls_2$$

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

Now, we can define

$$C_{post}(c', far) =$$
$$\{c : far \mid postcond(c) \; opposite \; postcond(c')\}$$
$$\cup$$
$$\{c : far \mid \exists x : post\_predicates(c); \; y : post\_predicates(c') \bullet x_{\Rightarrow} \; opposite \; y_{\Rightarrow}\}$$

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

5 Example: the Lift System

We first consider a simple lift with the following requirements:

1. The lift is called by pressing a button, either at a floor or inside the lift.
2. Pressing a call button is possible any time.
3. When the lift passes by floor k, and there is a call from this floor, then the lift will stop at floor k.
4. When the lift has stopped, it will open the door.
5. When the lift door has been opened, it will close automatically after d time units.
6. The lift only changes its direction when there are no more calls in the current direction.
7. When there are no calls, the lift stays at the floor last served, door closed.
8. As long as there are unserved calls, the lift will serve these calls.
9. When the lift is halted at floor k with the door open, a call for floor k is not taken into account.

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

11. Whenever the lift moves, its door must be closed.

Afterwards, we add the following features:

12. The closing of the door may be prevented by pressing an *open_door* button.

13. When something blocks the door, the lift interrupts the process of closing the door, and reopens the door.

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

15. The lift gives priority to calls from the executive landing.

In this paper, we will only show how Requirements 14 and 15 are added to the set of constraints, and how their interaction candidates are determined. Based on Requirements 1–13, we have the facts

1. The door can only be opened when it is closed or when it is closing and the door button is pressed.

2. When the door starts closing, it either will close completely, or closing is interrupted by pressing the door button.

3. The door button can only be pressed when it is released, and vice versa.

4. The door cannot be blocked when it is closed.

5.1 Starting Point

The following tables present the schematic constraints for the facts and for Requirements 1–13, and the corresponding tables of semantic relations. The formalized facts and Requirements 1–13 are given in Appendix B.

The schematic constraints (see Step 2 of the agenda of Section 3) are given in Table 1. Underlined parts show changements of constraints because of detected interactions. 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.

5.2 Adding new features

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

Requirement 14:

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

Step 1: Formalize the new constraint as a formula on system traces.

$$\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \bullet overloaded(tr(i).s) \Rightarrow door\_open(tr(i).s))$$

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

Constraint	schematic expression	Interaction with
$fact_1$	$open \rightsquigarrow end\_close$ $open \rightsquigarrow press\_door\_button$ $end\_close \rightsquigarrow open$ $press\_door\_button \rightsquigarrow open$	
$fact_2$	$begin\_close \rightsquigarrow end\_close$ $begin\_close \rightsquigarrow press\_door\_button$ $end\_close \rightsquigarrow begin\_close$ $press\_door\_button \rightsquigarrow begin\_close$	
$fact_3$	$press\_door\_button \rightsquigarrow release\_door\_button$	
	$release\_door\_button \rightsquigarrow press\_door\_button$	
$fact_4$	$door\_closed \rightsquigarrow \neg\ block$	
req_1	$press \wedge \underline{\neg\ at} \rightsquigarrow call$	
req_2	$true \rightsquigarrow press$	
req_3	$passes\_by \wedge call \rightsquigarrow stop$	
req_4	$stop \rightsquigarrow open$	
req_5	$open \rightsquigarrow begin\_close \underline{\vee\ press\_door\_button \vee\ block}$	
req_6	$direction = up \wedge call\_from\_up \rightsquigarrow direction = up$ $direction = down \wedge call\_from\_down \rightsquigarrow direction = down$	
req_7	$halted \wedge \neg\ call \rightsquigarrow halted$	
req_8	$call \wedge \neg\ at \rightsquigarrow at$	
req_9	$halted \wedge door\_open \wedge press \wedge at \wedge call \rightsquigarrow call$ $halted \wedge door\_open \wedge press \wedge at \wedge \neg\ call \rightsquigarrow \neg\ call$	req_1
req_{10}	$halted \wedge door\_closed \wedge press \wedge at \rightsquigarrow open$	
req_{11}	$\neg\ halted \Rightarrow door\_closed$	
$req_{12.a}$	$press\_door\_button \rightsquigarrow open\_requested$	
$req_{12.b}$	$open\_requested \wedge halted \rightsquigarrow door\_open$	req_5, req_8, req_{10}
req_{13}	$block \rightsquigarrow open$	req_5, req_8

Table 1: Overview of schematic constraints

$$\neg\ halted \rightsquigarrow stop \qquad door\_closed \rightsquigarrow open$$
$$halted \rightsquigarrow move \qquad door\_open \rightsquigarrow begin\_close$$
$$door\_closed \rightsquigarrow move \qquad \neg\ door\_closed \rightsquigarrow end\_close$$
$$\neg\ open\_requested \rightsquigarrow press\_door\_button \qquad \neg\ door\_closed \rightsquigarrow block$$

Table 2: Necessary conditions for events

Step 3: Update the tables of semantic relations. With this constraint, we have introduced a new predicate symbol *overloaded*. Hence, we must add the lines

$enter \rightsquigarrow overloaded$
$leave \rightsquigarrow \neg\ overloaded$

to Table 3. This in turn introduces two new events *enter* and *leave*, which causes us to add the lines

$door\_open \rightsquigarrow enter$
$door\_open \rightsquigarrow leave$

$press \rightsquigarrow call$	
$stop \rightsquigarrow \neg\ call$	$stop \rightsquigarrow at$
$move \rightsquigarrow passes\_by$	$move \rightsquigarrow \neg\ at$
$stop \rightsquigarrow \neg\ passes\_by$	$end\_close \rightsquigarrow door\_closed$
$press \rightsquigarrow call\_from\_up$	$open \rightsquigarrow \neg\ door\_closed$
$stop \rightsquigarrow \neg\ call\_from\_up$	$press\_door\_button \rightsquigarrow open\_requested$
$press \rightsquigarrow call\_from\_down$	$release\_door\_button \rightsquigarrow \neg\ open\_requested$
$stop \rightsquigarrow \neg\ call\_from\_down$	$open \rightsquigarrow door\_open$
$stop \rightsquigarrow halted$	$begin\_close \rightsquigarrow \neg\ door\_open$
$move \rightsquigarrow \neg\ halted$	

Table 3: Events establishing predicate literals

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

Table 4: Relations between predicate literals

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

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

According to the equivalences

$$p \in q_{\Rightarrow} \ \Leftrightarrow\ q \in {}_{\Rightarrow}p \ \Leftrightarrow\ \neg\ q \in \neg\ p_{\Rightarrow}$$

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

$$
\begin{aligned}
\neg\ door\_open_{\Rightarrow} &= \{\neg\ overloaded\} \\
door\_closed_{\Rightarrow} &= \{\neg\ door\_open, \neg\ overloaded\} \\
\neg\ halted_{\Rightarrow} &= \{passes\_by, \neg\ at, door\_closed, \neg\ door\_open, \neg\ overloaded\} \\
passes\_by_{\Rightarrow} &= \{\neg\ at, \neg\ halted, door\_closed, \neg\ door\_open, \neg\ overloaded\}
\end{aligned}
$$

Note that we do not change the entry for $\neg\ at_{\Rightarrow}$, because *at* has a floor as its argument. If $\neg\ at(f)$ holds, we do not know if the lift is moving or if it is at another floor than f. The above

equivalences are exact only for predicates without arguments (e.g., $door\_closed, halted$). For the other predicates, they just point out which entries of the table must be re-considered.

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:

$$\begin{aligned} pre\_predicates(req_{14}) &= \{overloaded\} \\ \Rightarrow overloaded \cup overloaded_{\Rightarrow} &= \{door\_open, \neg\ door\_closed, halted, at, \\ &\quad \neg\ passes\_by\} \\ \{e : EVENT;\ y : \Rightarrow overloaded \cup overloaded_{\Rightarrow} & \\ \mid y \rightsquigarrow e \bullet e\} &= \{open, move\} \end{aligned}$$

Hence, the precondition interaction candidates are the ones that have one of the elements $door\_open, \neg\ door\_closed, halted, at, \neg\ passes\_by, open, move$ in their precondition. According to Table 1, these are $fact_1$, req_5, req_7, req_9, req_{10}.

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

$$post\_predicates(req_{14}) = \{\neg\ door\_open\}$$

Because $door\_open_{\Rightarrow} = \{\neg\ door\_closed, \neg\ passes\_by, halted, at\}$, we must look for postconditions $door\_closed, passes\_by, \neg\ halted, \neg\ at$ and related events according to Table 3. These are $end\_close, move$. According to Table 1, we get the candidates $fact_1$ and req_{11}.

Step 5: Analyze possible interactions. We do not have interactions with $fact_1$, req_7, req_9, req_{10}, req_{11}, but with req_5. There is also an interaction with req_8, which cannot be detected by our procedure because of missing semantic information. In Section 6, we discuss in more detail why this interaction cannot be found and what can be done about this.

Step 6: Eliminate interactions, if necessary. To adjust req_5 (see Appendix B), we cannot use the macro $must\_be\_followed\_by_s$ any more, because now we do not add a new possible event, but a predicate. We must expand the macro and add the postcondition

$$\ldots \vee \exists j : i + 1 \,..\, \#tr \bullet overloaded(tr(j).s)$$

The new schematic constraint becomes

$$open \rightsquigarrow begin\_close \vee press\_door\_button \vee block \vee overloaded$$

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 $begin\_close, end\_close, move$. In Table 1, we find the candidates $fact_1$, $fact_2$, and req_{11}. There is no interaction with any of them.

To adjust req_8, we add the elements $enter$ and $leave$ to the set evs (see Appendix B). Its schematic version remains the same. Hence, no further interaction analysis is necessary.

Requirement 15:

The lift gives priority to calls from the executive landing.

Step 1: Formalize the new constraint as a formula on system traces.

$$\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \bullet call(tr(i).s, executive\_floor) \\ \Rightarrow next\_stop(tr(i).s) = executive\_floor)$$

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

Step 3: Update the tables of semantic relations. We did not introduce new predicates or events, only a new function symbol *next_stop* and a constant of type *Floor*. Hence, the semantic tables remain unchanged.

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:

$$\begin{aligned} pre\_predicates(req_{15}) &= \{call\} \\ {}_{\Rightarrow}call \cup call_{\Rightarrow} &= \{call\_from\_up, call\_from\_down\} \\ \{e : EVENT;\ y : {}_{\Rightarrow}call \cup call_{\Rightarrow} \mid y \rightsquigarrow e \bullet e\} &= \varnothing \end{aligned}$$

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_3, req_6, req_8, req_9.

There cannot be any postcondition interaction candidates, because the postcondition of req_{15} contains only new syntactic elements that are not semantically related to any of the other syntactic elements.

Step 5: Analyze possible interactions. We have interactions with req_3 and req_6, but not with req_8 and req_9.

Step 6: Eliminate interactions, if necessary. We add a new precondition to req_3, which becomes

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

The new schema for req_3 is:
$passes\_by \wedge call \wedge f = executive\_floor \rightsquigarrow stop$
$passes\_by \wedge call \wedge f \neq executive\_floor \wedge \neg\ call \rightsquigarrow stop$

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*. We have ${}_{\Rightarrow}\neg\ call \cup \neg\ call_{\Rightarrow} = \{\neg\ call\_from\_up, \neg\ call\_from\_down\}$. Because there are no related events, our candidates are the constraints with precondition $\neg$ *call*, $\neg$ *call_from_up*, $\neg$ *call_from_down*. These are req_7 and req_9. With both of them, there is no interaction.

To adjust req_6, we also add new preconditions.

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

The new schemas 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_3, 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_3. 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.

Detecting more interactions. In Section 5.2, we saw that our procedure did not find req_8 as an interaction candidate for req_{14} although there is an interaction between these requirements. The reason is that our tables did not contain enough information to detect this interaction. Our constraints do not say what the lift has to do to get to a certain floor when it is elsewhere. Human analysts detect the interaction only because they know how a lift works. If the lift is moving, it must stop at the requested floor. It it is halted with the door closed, it must start moving. If it is halted with the door open, it must close the door and then start moving. Only in this last case there is an interaction with req_{14}, which requires the door to be opened.

Clearly, an automatic procedure can only work if it is given enough information. Such information, however, can be added systematically. The liveness condition req_8 is distinguished from other constraints such as req_1, req_{14} and req_{15} by the fact that it relates states of the system that can be separated by a large number of events. In contrast, req_1 relates consecutive states, and req_{14} and req_{15} talk about one state only.

Accordingly, constraints can be assigned a *distance*, which characterizes the different states related by the constraint. Requirement req_1 would have distance one, req_{14} and req_{15} would

have distance zero, and req_8 would have a distance greater than one. For each constraint with a distance distance greater than one, additional information is needed. Such information can be expressed as scenarios that show on the one hand how to proceed one step from the beginning state (to perform analysis of precondition interaction) and on the other hand one step that leads to the final state (to perform analysis of postcondition interaction). For req_8, this would yield the scenarios

$$call \wedge \neg\ at \wedge \neg\ halted \rightsquigarrow stop$$
$$call \wedge \neg\ at \wedge halted \wedge door\_closed \rightsquigarrow move$$
$$call \wedge \neg\ at \wedge halted \wedge door\_open \rightsquigarrow begin\_close$$

for the precondition analysis and the scenario

$$stop \rightsquigarrow at$$

for the postcondition analysis. When such scenarios are added to the sets of constraints, our procedure finds the interaction between req_{14} and req_8 via the common precondition *halted*.

On the other hand, those interactions that are not detected by analysis of the requirements should become apparent and be resolved when the formal specification is set up. Our approach leaves room for decisions how important an early detection of interactions is considered to be and how much effort is spent for this activity. If an early detection of interactions is important, then our procedure can be adjusted. If it is acceptable to detect some interactions only in the specification phase, then a simpler procedure can be used in the requirements elicitation phase.

References

[Hei98] Maritta 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] Maritta Heisel and Jeanine Souquières. Methodological support for requirements elicitation and formal specification. In A. Finkelstein, editor, *Proceedings 9th International Workshop on Software Specification and Design*, 1998. to appear.

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

A Formal Expression of Constraints on Traces

We express requirements, assumptions, and facts referring to the current state of the system, events that happen, and the time an event happens:

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

The system is started in state S_1. When event e_1 happens at t_1, then the system enters state S_2, and so forth. One element of a trace of the system thus consists of these three parts. 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 at an earlier time as an event 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.

$$TRACE : \mathbb{P}(\text{seq } TraceItem)$$

$$\forall tr : TRACE \bullet \forall i : \text{dom } tr \bullet i = \#tr \vee (tr\ i).t \leq_t (tr(i+1)).t$$

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

$$Tr : \mathbb{P}\ TRACE$$

$$\forall tr : Tr \bullet (\forall tr' : TRACE \mid tr' \text{ prefix } tr \bullet tr' \in Tr)$$

To express the constraints, it will also be necessary to declare predicates on the states, because the behavior of the system may depend on its current state. Such predicates, however, are only *declared* in the requirements elicitation phase. Their *definition* is part of the specification phase. But also predicates that refer to the occurrence of events at certain points in time are conceivable.

A.1 Specification Macros for Traces

To express constraints concisely, we define several specification macros.

Often, it is necessary to select subtraces tr' of a given trace tr that begin with an event e_1 and end when the event e_2 occurs for the first time after e_1 has occurred:

$$subtraces : TRACE \times EVENT \times EVENT \longrightarrow \mathbb{P}\ TRACE$$

$$\begin{array}{l}
\forall tr, tr' : TRACE;\ e_1, e_2 : EVENT \bullet \\
\quad tr' \in subtraces(tr, e_1, e_2) \\
\quad \Leftrightarrow \\
\quad (\exists tr_1, tr_2 : TRACE \bullet tr = tr_1 \frown tr' \frown tr_2) \wedge \\
\quad (tr'\ 1).e = e_1 \wedge (tr'\ (\#tr)).e = e_2 \wedge \\
\quad (\forall i : 2 \,..\, \#tr' - 1 \bullet (tr'\ i).e \neq e_2)
\end{array}$$

The macro *alternates_with* expresses that events e_1 and e_2 must always alternate.

$$\_alternates\_with\_ : EVENT \leftrightarrow EVENT$$

$$\begin{array}{l}
\forall e_1, e_2 : EVENT \bullet \\
\quad e_1\ alternates_with\ e_2 \\
\quad \Leftrightarrow \\
\quad (\forall tr : Tr \bullet (\forall tr' : subtraces(tr, e_1, e_1) \bullet \\
\qquad (\exists_1 ti : \text{ran } tr' \bullet ti.e = e_2)))
\end{array}$$

A generalization of *alternates_with* is the following:

$$\_alternates\_with_s\_ : EVENT \leftrightarrow \mathbb{P}\ EVENT$$

$$\begin{array}{l}
\forall ev : EVENT;\ evs : \mathbb{P}\ EVENT \bullet \\
\quad ev\ alternates_with_s\ evs \\
\quad \Leftrightarrow \\
\quad (\forall tr : Tr \bullet (\forall tr' : subtraces(tr, ev, ev) \bullet \\
\qquad \exists_1 ev' : evs \bullet \exists ti : \text{ran } tr' \bullet ti.e = ev'))
\end{array}$$

Here, event ev must alternate with the events contained in the set of events evs.

The next macro expresses that event e_1 must be immediately followed by event e_2.

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

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

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

The next macro expresses that after event e_1 has happened, event e_2 is possible.

$$
\begin{array}{|l}
\_may\_be\_followed\_by\_ : EVENT \leftrightarrow EVENT \\
\hline
\forall\, e_1, e_2 : EVENT \bullet \\
\quad e_1\ may\_be\_followed\_by\ e_2 \\
\quad \Leftrightarrow \\
\quad (\forall\, tr : Tr \mid (last\ tr).e = e_1 \bullet \\
\qquad (\exists\, tr' : Tr \mid tr\ \mathsf{prefix}\ tr' \bullet \\
\qquad\quad (\exists\, i : \mathrm{dom}\ tr' \mid i > \#tr \bullet (tr\ i).e = e_2)))
\end{array}
$$

The next macro expresses that if event e_1 happens, then event e_2 must happen within d time units.

$$
\begin{array}{|l}
\_must\_be\_followed\_by\_ : EVENT \leftrightarrow (EVENT \times TIME) \\
\hline
\forall\, e_1, e_2 : EVENT; d : TIME \bullet \\
\quad e_1\ must\_be\_followed\_by\ (e_2, d) \\
\quad \Leftrightarrow \\
\quad (\forall\, tr : Tr \bullet \forall\, i : \mathrm{dom}\ tr \mid tr(i).e = e_1 \land tr(\#tr).t - tr(i).t \geq d \bullet \\
\qquad (\exists\, j : i + 1\,..\,\#tr \bullet tr(j).e = e_2 \land tr(j).t - tr(i).t \leq d))
\end{array}
$$

A generalization of this macro is:

$$
\begin{array}{|l}
\_must\_be\_followed\_by_s\_ : EVENT \leftrightarrow (\mathbb{P}\ EVENT \times TIME) \\
\hline
\forall\, ev : EVENT; evs : \mathbb{P}\ EVENT; d : TIME \bullet \\
\quad ev\ must\_be\_followed\_by_s\ (evs, d) \\
\quad \Leftrightarrow \\
\quad evs \neq \varnothing \land \\
\quad (\forall\, tr : Tr \bullet \forall\, i : \mathrm{dom}\ tr \mid tr(i).e = e_1 \land tr(\#tr).t - tr(i).t \geq d \bullet \\
\qquad (\exists\, j : i + 1\,..\,\#tr \bullet tr(j).e \in evs \land tr(j).t - tr(i).t \leq d))
\end{array}
$$

A.2 Auxiliary functions

The function *events* transforms a trace into a sequence of events.

$$events : TRACE \longrightarrow \operatorname{seq} EVENT$$

$$\forall\, tr : TRACE \bullet events\ tr = \{i : \mathbb{N};\ ti : TraceItem \mid i \mapsto ti \in tr \bullet i \mapsto ti.e\}$$

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

$$remove : TRACE \times \mathbb{P}\, EVENT \longrightarrow TRACE$$

$$\forall\, tr : TRACE;\ evs : \mathbb{P}\, EVENT \bullet remove(tr, evs) = tr \upharpoonright \{ti : TraceItem \mid ti.e \notin evs\}$$

B Formal Versions of Requirements and Facts

The basics of this formalization are given in Appendix A.

Fact 1:

The door can only be opened when it is closed or when it is closing and the door button is pressed.

$$open\ alternates\_with_s\ \{end\_close, press\_door\_button\}$$

Fact 2:

When the door starts closing, it either will close completely, or closing is interrupted by pressing the door button.

$$begin\_close\ alternates\_with_s\ \{end\_close, press\_door\_button\}$$

Fact 3:

The door button can only be pressed when it is released, and vice versa.

$$press\_door\_button\ alternates\_with\ release\_door\_button$$

Fact 4:

The door cannot be blocked when it is closed.

$$\forall\, tr : Tr \bullet \forall\, i : \operatorname{dom} tr \mid i \neq \#tr \bullet$$
$$door\_closed(tr(i).s) \Rightarrow tr'(i+1).e \neq block)$$

Requirement 1:

The lift is called by pressing a button, either at a floor or inside the the lift.

$$\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr;\ b : Button \mid i \neq \#tr \bullet \\ tr(i).e = press(b) \land \neg\ at(tr(i).s, floor(b)) \Rightarrow call(tr(i+1).s, floor(b))$$

Requirement 2:

Pressing a call button is possible any time.

$$\forall\, tr : Tr;\ et_1, et_2 : \mathrm{seq}\, EVENT;\ b : Button \mid events\, tr = et_1 \frown et_2 \bullet \\ \exists\, tr' : Tr \bullet events\, tr' = et_1 \frown \langle press(b) \rangle \frown et_1$$

where the definition of the function *events* can be found in Appendix A.2.

Requirement 3:

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

$$\forall\, tr : Tr \bullet (\mathbf{let}\ tr' == remove(tr, \{b : Button \bullet press(b)\}) \bullet \\ \forall\, i : \mathrm{dom}\, tr';\ k : Floor \mid i \neq \#tr' \bullet \\ passes\_by(tr'(i).s, k) \land call(tr'(i).s, k) \Rightarrow tr'(i+1).e = stop(k))$$

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

Requirement 4:

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

$$\forall\, tr : Tr \bullet (\mathbf{let}\ tr' == remove(tr, \{b : Button \bullet press(b)\}) \bullet \\ \forall\, i : \mathrm{dom}\, tr' \mid i \neq \#tr' \bullet \\ tr'(i).e = stop(k) \Rightarrow tr'(i+1).e = open)$$

Requirement 5:

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

$$open\ must\_be\_followed\_by_s\ (\{begin\_close, press\_door\_button, block\}, d)$$

Requirement 6:

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

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

Requirement 7:

When there are no calls, the lift stays at the floor last served, door closed.

$$\begin{array}{l}\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \mid i \neq \#tr \bullet \\ \quad halted(tr(i).s) \land (\forall\, k : Floor \bullet \neg\, call(tr(i).s, k)) \Rightarrow halted(tr(i+1).s)\end{array}$$

That the door is closed follows already from Requirement 5. A redundant formulation of req_7 would be $\ldots \Rightarrow halted(tr(i+1).s) \land door\_closed(tr(i+1).s)$.

Requirement 8:

As long as there are unserved calls, the lift will serve these calls.

We cannot require that when the lift is halted and receives a call, it starts moving immediately. For example, when the lift just has arrived, its door is still closed. It must then open the door to let passengers enter or leave, and it must close the door again, before it can serve the new call. Hence, we introduce a constant c that represents the maximal number of events that may happen before the lift arrives at the requested floor.

To express Requirement 8 formally, we consider traces tr where at some point there is a call for a floor f, but the lift is not at floor f. We are only interested in the part tr_2 of the trace that begins with such a state. The subtrace tr_2 must be long enough, i.e., it must contain at least c events that are not *press*, *press_door_button*, *release_door_button* or *block* events. These are the events that may prevent the lift to serve a call for an unlimited amount of time. We then require that there must be a state in tr_2 where the lift at the requested floor f, such that no more than c events have happened that do not delay the lift.

$$\begin{array}{l}\textbf{let}\ evs == \{b : Button \bullet press(b)\} \cup \{press\_door\_button, release\_door\_button, block\} \bullet \\ \quad (\forall\, tr : Tr \bullet \forall\, tr_1, tr_2 : TRACE \mid tr = tr_1 \frown tr_2 \land \#(remove(tr_2, evs)) > c \bullet \\ \qquad \forall f : Floor \bullet call(tr_2(1).s, f) \land \neg\, at(tr_2(1).s, f) \Rightarrow \\ \qquad\quad \exists\, tr_3, tr_4 : TRACE \mid tr_2 = tr_3 \frown tr_4 \land tr_4 \neq \langle\rangle \bullet \\ \qquad\qquad at(tr_4(1).s, f) \land \#(remove(tr_3), evs) \leq c)\end{array}$$

Requirement 9:

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

$$\begin{array}{l}\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr;\ b : Button \mid i \neq \#tr \bullet \\ \quad halted(tr(i).s) \land door\_open(tr(i).s) \land tr(i).e = press(b) \land at(tr(i).s, floor(b)) \\ \qquad \Rightarrow \forall f : Floor \bullet call(tr(i+1).s, f) \Leftrightarrow call(tr(i).s, f)\end{array}$$

Requirement 10:

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

As for all liveness requirements, we must express this constraint without taking into account any delaying events.

$$\begin{array}{l}\textbf{let } evs == \{b : Button \bullet press(b)\} \cup \{press\_door\_button, release\_door\_button\} \bullet \\ \quad (\forall\, tr : Tr \bullet \forall\, tr_1, tr_2 : TRACE \mid tr = tr_1 \frown tr_2 \wedge remove(tr_2, evs) \neq \langle\rangle \bullet \\ \qquad halted(tr_2(1).s) \wedge door\_closed(tr_2(1).s) \wedge tr_2(1).e = press(b) \wedge at(tr_2(1).s, floor(b)) \\ \qquad\quad \Rightarrow remove(tr_2, evs)(1).e = open)\end{array}$$

Requirement 11:

Whenever the lift moves, its door must be closed.

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

Requirement 12:

The closing of the door may be prevented by pressing an *open_door* button.

This requirement forces us to have two events *begin_close* and *end_close* instead of one event *close*, because events are instantenous and cannot be interrupted.

We split the requirement into two requirements $req_{12.a}, req_{12.b}$

$$\begin{array}{l}\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \mid i \neq \#tr \bullet \\ \quad tr(i).e = press\_door\_button \Rightarrow open\_requested(tr(i+1).s)\end{array}$$

$$\begin{array}{l}\forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \mid i \neq \#tr \bullet \\ \quad open\_requested(tr(i).s) \wedge halted(tr(i).s) \Rightarrow door\_open(tr(i+1).s)\end{array}$$

Requirement 13:

When something blocks the door, the lift interrupts the process of closing the door, and reopens the door.

$$\begin{array}{l}\textbf{let } evs == \{b : Button \bullet press(b)\} \cup \{press\_door\_button, release\_door\_button\} \bullet \\ \quad (\forall\, tr : Tr \bullet (\textbf{let } tr' == remove(tr, evs) \bullet \\ \qquad \forall\, i : \mathrm{dom}\, tr' \mid i \neq \#tr' \bullet \\ \qquad\quad tr'(i).e = block \Rightarrow tr'(i+1).e = open))\end{array}$$