A Heuristic Algorithm to Detect Feature Interactions in Requirements

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 heuristic algorithm to systematically detect feature interactions in requirements, which are expressed as constraints on system event traces. The algorithm is part of a broader methodology for requirements elicitation and formal specification. Given a new constraint and a set of already accepted constraints, it computes a set of candidate constraints that possibly interact with the new one. We illustrate the algorithm by adding new features to a simple lift.

1 Introduction

The term "feature" has been coined in telecommunications, where a feature is some service a client may subscribe to, such as call forwarding. It was also in telecommunications where the problem of feature interaction occurred first.

Nowadays, features and the problem of integrating them are no longer confined to the area of telecommunications. Features can be identified in almost every software system, and there are even proposals to base the whole software engineering process on features [3].

Following Turner et al. [3], we consider a feature as "a coherent and identifiable bundle of system functionality that helps characterise the system from the user perspective." More technically, a feature is "a clustering or modularisation of individual requirements" of a requirements specification. This definition emphasises the user-oriented nature of features.

When a system is described in terms of several features, the situation may occur where the features make perfect sense when considered in isolation, but their combination leads to contradictions or unwanted or unexpected system behaviour. This situation is called *feature interaction.*

Although some authors (e.g., [4,5]) distinguish between "good" and "bad" interactions, we prefer not to do so, because the question whether an interaction is desirable or not must be decided by the user and must only be addressed when the *integration* of the various features is undertaken.

Different approaches to feature-oriented software development can be distinguished according to the following criteria:

1. How are features represented?
2. How are feature interactions detected/avoided?
3. How are feature interactions resolved?
4. How are features composed/integrated?

In our work, these questions are answered as follows:

1. Features are sets of formalised requirements, where each requirement is represented as a formula.
2. Because requirements are elicited in collaboration with the users by a brainstorming process, we do not attempt to avoid interactions right from the beginning but to detect and resolve them as early as possible. How we achieve this goal is the topic of this paper.
3. We do not attempt to resolve feature interactions automatically, because we think that each interaction should be discussed with the clients. It is then up to the clients to decide on the required system behaviour.
4. Feature composition corresponds to conjunction of formulae. Hence, it is commutative and associative. We think that these properties are desirable, because, for example, the behaviour of a telephone system should not depend on the order the user subscribed to the different features.

In this paper—which is a revised and extended version of the position paper [6]—we present an algorithm that, given a set of already accepted requirements and a new requirement to be added, calculates a set of candidate requirements with whom there might be an interaction. The algorithm is *heuristic*, which means that we cannot guarantee that all existing interactions are indeed detected.[1] It was developed as part of a method for requirements engineering, but it is also useful for the evolution of systems.

Section 2 gives a brief overview of our requirements elicitation method. Section 3 describes how to incorporate a single constraint into a set of existing constraints. Section 4 presents the algorithm for calculating interaction candidates. Section 5 illustrates the application of the approach by way of adding new features to a simple lift system. Related work is discussed in Section 6. Section 7 concludes the paper with a discussion of the approach and its benefits.

2 Method for Requirements Elicitation

Our method for requirements engineering [7,8] 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 a formal specification. 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.

Our approach to requirements engineering is inspired by the work of Jackson and Zave [9,10] and by the first steps of object oriented methods and

[1] Striving for a provably correct and complete algorithm would necessitate a formal and decidable notion of interaction. Because the notion of interaction covers more phenomena than just logical inconsistency, it is questionable if such a definition is possible or even desirable.

notations such as UML [11]. The starting point is a brainstorming process where the application domain and the requirements are described in natural language. This informal description is then transformed into a formal representation. Requirements elicitation is performed in five steps:

1. Introduce the domain vocabulary.
   The different notions of the application domain are expressed in a textual or graphical 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 behaviour.
   It does not suffice to just state requirements for the system. Often, facts and assumptions must be introduced to make the requirements satisfiable. *Facts* express conditions that always hold in the application domain, regardless of the implementation of the software system. Other requirements cannot be enforced, because e.g., human users might violate regulations. 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. For each event, it must be stated who is in control of the event (the software system or its environment) and who can observe it.
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.
5. Formalise the facts, assumptions, and requirements as constraints on the possible traces of system events.

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

- It is possible to express *negative* requirements, i.e., to require that certain things do not happen.
- It is possible to give scenarios, i.e., examples of system behaviour.
- Giving constraints does not fix the system behaviour entirely. The specification is not restricted unnecessarily. Any specification that fulfils the constraints is admitted [7].

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, stating necessary semantic conditions that the developed artifact must fulfil in order to serve its purpose properly:

- 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 Method to Incorporate Single Constraints

In Step 5 of the method, facts, assumptions, and requirements must be formalised one by one. But before a new formalised constraint is added to the set of already accepted constraints, its possible interactions with them should be analysed, in order to detect inconsistencies or undesired behaviour.

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

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

5.1 Formalise the new constraint as a formula on system traces.
To formalise 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 Tr be the set of possible traces. A constraint is expressed as a formula restricting the set Tr. For a given trace $tr \in Tr$, $tr(i)$ denotes the i-th element of this trace, $tr(i).s$ the state of the i-th element, $tr(i).e$ the event which occurs in that state, and $tr(i).t$ is the time at which the event occurs. For each possible trace, its prefixes are also possible traces. A formal specification of traces is given in Appendix A.
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 stated. These events must be shared with the software system.
If possible, we recommend expressing 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.

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

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

This formula contains the event symbol $press(b)$, which is parameterised with a button, and the predicate symbols *halted*, *at*, *door_open* and *call*. The predicates *at* and *call* have a floor as an additional parameter besides the system state. The function *floor* associates a floor with a button. The expression dom tr denotes the valid indices of the trace, i.e., $1\,..\,\#tr$, where $\#$ denotes the length of a trace.

5.2 Give a schematic expression of the constraint.
Our algorithm to determine interaction candidates uses schematic versions of formalised constraints. These have the form

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

where the x_i, y_j are literals and $\diamond$ 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.

Example. The schematic expression corresponding to the constraint stated before is

$$\mathit{halted} \wedge \mathit{at} \wedge \mathit{door\_open} \wedge \mathit{press} \rightsquigarrow \neg\, \mathit{call}$$

5.3 Update the tables of semantic relations.
Because our algorithm is completely automatic, it cannot be based on syntax alone. We also must take into account the semantic relations between the different symbols. We construct three tables of semantic relations:

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

   Example. $\mathit{door\_open}_{\Rightarrow} = \{\neg\, \mathit{door\_closed}, \mathit{halted}, \mathit{at}, \neg\, \mathit{passes\_by}\}$

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

5.4 Determine interaction candidates, based on the list of schematic requirements (Step 5.2) and the semantic relation tables (Step 5.3).
The definition of the interaction candidates is given in Section 4.

5.5 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 prove that an interaction exists between the new constraint and each candidate. It is up to the analyst and the customer to decide if the conjunction of the new constraint with the candidates yields an unwanted behaviour or if it even is contradictory.

5.6 Resolve interactions.
To resolve an interaction, we usually relax requirements or strengthen

assumptions. Once a constraint has been modified, an interaction analysis on those literals that were changed or newly introduced must be performed.

The following validation conditions are associated with Step 5 of the method for requirements elicitation:

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

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

4 Determining Interaction Candidates

Two constraints are interaction candidates for one another if they have overlapping preconditions but incompatible postconditions, as is illustrated in Figure 1. "Incompatible" does not necessarily mean "logically inconsistent"; it could also mean "inadequate" for the purpose of the system.

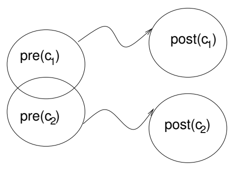

Fig. 1. Interaction candidates

Our algorithm 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 the constraints with incompatible postconditions. If in such a case the preconditions do not exclude each other, an interaction occurs.

4.1 Precondition Interaction Candidates

If two constraints[2] $\underline{x} \leadsto \underline{y}$ and $\underline{u} \leadsto \underline{w}$ have common literals in their precondition ($\underline{x} \cap \underline{u} \neq \varnothing$), then they are certainly interaction candidates.

[2] Underlined identifiers denote sets of literals.

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

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

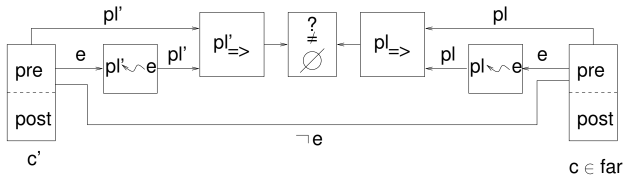

Fig. 2. Determining interaction candidates by precondition analysis

Figure 2 shows how to calculate interaction candidates $C_{pre}(c', far)$ by a precondition analysis for a new constraint c' with respect to the set far of facts, assumptions, and requirements already defined. Let

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

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

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

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

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

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

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

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

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

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

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

4.2 Postcondition Interaction Candidates

To find conflicting postconditions, we compute the predicative closure of the postcondition of the new constraint c' and the one of each constraint $c \in \mathit{far}$ in much the same way as for the preconditions. For an event e contained in the postcondition of a constraint, all predicate literals pl with $e \leadsto pl$ belong to the set $\mathit{post\_predicates}(c)$:

$$\begin{array}{l}
\mathit{postcond}(x_1 \diamond x_2 \diamond \ldots \diamond x_n \leadsto y_1 \diamond y_2 \diamond \ldots \diamond y_k) = \{y_1, \ldots y_k\} \\
e_{\leadsto} = \{pl : \mathit{PLit} \mid e \leadsto pl\} \\
\mathit{post\_predicates}(c) = (\mathit{postcond}(c) \cap \mathit{PLit}) \cup \bigcup_{e \in \mathit{postcond}(c) \cap \mathit{EVENT}} e_{\leadsto}
\end{array}$$

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

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

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

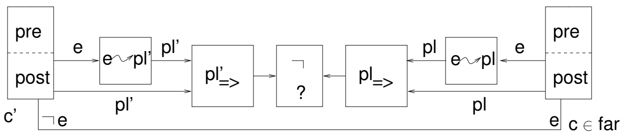

Fig. 3. Determining interaction candidates by postcondition analysis

Again, the two cases are necessary, because postconditions may contain event literals, whereas predicative closures only contain predicate literals.

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

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

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

As already mentioned, computing the set $C(c', \mathit{far})$ can be performed completely automatically. Moreover, the implementation is fairly simple and efficient. No theorem proving techniques or other search strategies are necessary.

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 at 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 formalise 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

Tables 1–4 present the schematic constraints for the fact and Requirements 1–10, and the corresponding tables of semantic relations. The formalised fact and Requirements 1–10 are given in Appendix B.

The schematic constraints, see Step 5.2 of the method 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 detection, however, requires the precondition to refer to an earlier or the same state as the postcondition. Hence, the schematic expressions 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 $P \Rightarrow Q$).

The events establishing the predicates and their negations are given in Table 2, ordered alphabetically with respect to the predicate symbols. Table 3 shows the necessary conditions for the events, ordered alphabetically with respect to the event symbols. Finally, Table 4 shows the relations between the various predicate literals. This information is collected when performing Step 5.3 of the method 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

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

Table 2. Events establishing predicate literals

5.2 Adding new features

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

$door\_open \leftsquigarrow close$	$halted \leftsquigarrow move$
$at \leftsquigarrow move$	$door\_closed \leftsquigarrow open$
$call \leftsquigarrow move$	$\neg\ halted \leftsquigarrow stop$
$door\_closed \leftsquigarrow move$	$passes\_by \leftsquigarrow stop$

Table 3. Necessary conditions for events

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

Table 4. Relations between predicate literals

Requirement 11 (the *Overloaded* Feature). When the lift is overloaded, the door will not close. Some passengers must get out.

Step 5.1: Formalise 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 5.2: Give a schematic expression of the constraint.
$overloaded \rightsquigarrow door\_open$

Step 5.3: 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

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

to Table 2 and the lines

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

to Table 3. To Table 4, we add the lines

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

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

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

Step 5.4: 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 of $overloaded_{\Rightarrow}$ in their precondition, i.e., *at*, *door_open*, $\neg$ *door_closed*, *halted*, $\neg$ *passes_by*. According to Table 1, these are *fact* because of $\neg$ *door_closed*, req_3 because of *at*, req_8 because of *at*, *halted* and *door_open*, req_9 because of *at* and *halted*. Requirement req_2 is always a candidate for precondition interaction, because *true* is implied by every predicate.

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

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

Because $door\_open_{\Rightarrow} = \{at, \neg\, door\_closed, halted, \neg\, passes\_by\}$, we must look for postconditions that contain one of the elements of predicates $\neg$ *at*, *door_closed*, $\neg$ *halted*, *passes_by* and related events that establish those predicates according to Table 2. These are *close* and *move*. According to Table 1, we get the candidates req_6 because of the event *close* and req_{10} because of *door_closed*.

Step 5.5: Analyse possible interactions.
We do not have interactions with *fact*, req_2, req_3, 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 5.6: Eliminate interactions, if necessary. We relax req_6 as follows:

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

The informal requirement req_6 has to be updated now. It becomes: “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 of req_6 becomes $open \rightsquigarrow close \vee overloaded$.

Since we have added the new postcondition *overloaded* to the constraint, we must now perform postcondition interaction analysis on this literal. With

$overloaded_{\Rightarrow} = \{at, door\_open, \neg\ door\_closed, halted, \neg\ passes\_by\}$ it follows that we must look for constraints which contain one of the predicates $\neg\ at$, $\neg\ door\_open$, $door\_closed, \neg\ halted, passes\_by$ in their postconditions and related events according to Table 2. These are *close* and *move*. In Table 1, we find the candidate req_{10}. There is no interaction with it.

This concludes the introduction of the *overloaded* feature. To add this feature to the lift, we not only had to introduce some new predicates and change some requirements of the base system. More importantly, we had to introduce two new events *enter* and *leave*. Our method requires that these events be observable by the software system. Hence, a weight sensor must be added to the lift if it is not already available.

Requirement 12 (the *Executive Floor* Feature). The lift gives priority to calls from the executive landing.

Step 5.1: Formalise the new constraint as a formula on system traces.

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

where $executive\_floor$ is a constant of type $FLOOR$.

Step 5.2: Give a schematic expression of the constraint.
$call \leadsto next\_stop\_at\_executive\_floor$

Step 5.3: Update the tables of semantic relations.
With this constraint, we have introduced a new predicate symbol $next\_stop\_at\_executive\_floor$ for which we must specify the events that modify it. We add the lines
$press \leadsto next\_stop\_at\_executive\_floor$
$stop \leadsto \neg\ next\_stop\_at\_executive\_floor$
to Table 2. We add the following entry to Table 4:
$next\_stop\_at\_executive\_floor_{\Rightarrow} = \{call\}$

Step 5.4: 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.

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

$$post\_predicates(req_{12}) = \{next\_stop\_at\_executive\_floor\}$$

Because $next\_stop\_at\_executive\_floor_{\Rightarrow} = \{call\}$, we must look for postconditions that contain the predicate $\neg\ call$ and related events according to Table 2, that is *stop*. According to Table 1, we get as candidates req_1, req_3 and req_8 because of $\neg\ call$ and req_4 because of the event *stop*.

Step 5.5: Analyse possible interactions.
We have no interactions with req_1, req_3 and req_8, but with req_4 and req_7, 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 expressed in req_7.

Step 5.6: Eliminate interactions, if necessary.
To adjust req_4, we add a new precondition to it; req_4 becomes

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

The informal requirement req_4 has to be updated. It becomes: "When the lift passes by a floor f, and there is a call from this floor, then the lift will stop at this floor if f is the executive floor or there is no call from the executive floor".

The new schematic expression for req_4 is:

$$passes\_by \wedge call \wedge (passes\_by\_executive\_floor \vee \neg\ call) \leadsto stop$$

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

Moreover, to capture the new precondition $f = executive\_floor$, we have introduced a new predicate symbol *passes_by_executive_floor* with

$$\begin{array}{l}
next\_stop\_at\_executive\_floor_{\Rightarrow} = \\
\quad \{passes\_by, \neg\ at, door\_closed, \neg\ door\_open, \neg\ halted\}
\end{array}$$

We must now perform a precondition interaction analysis on the new preconditions $\neg$ *call* and *passes_by_executive_floor*. Concerning $\neg$ *call*, our candidates are the constraints with precondition $\neg$ *call*, $\neg$ *call_from_up*, $\neg$ *call_from_down*, because there are no related events. There are no interaction candidates. Concerning *passes_by_executive_floor*, we also do not get any new candidates, because all candidates were already candidates because of *passes_by*.

To adjust req_7, we also add new preconditions.

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

The informal requirement req_7 has to be updated. It becomes: "The lift only changes its direction when there are no more calls in the current direction or there is a call from the executive floor".

The new schematic expressions for req_7 are:

$$\begin{array}{l}
direction\_up \wedge call\_from\_up \wedge \neg\ call \leadsto direction\_up \\
direction\_down \wedge call\_from\_down \wedge \neg\ call \leadsto direction\_down
\end{array}$$

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 Related work

In general, there are two ways to deal with the feature interaction problem. The first way is to *prevent* feature interactions right from the beginning, for example by enforcing modularity in the design of features. This approach is advocated by Jackson and Zave in their Distributed Feature Composition (DFC) virtual architecture [12]. Preventing feature interactions is supported by making feature first-class citizens in specification languages. For example, Plath and Ryan add a feature construct to the SMV language [13].

The second way to deal with feature interactions is to *detect* interactions and then resolve them. Even when the goal is to prevent feature interactions, algorithms for detecting them are indispensable. Zave [5] presents a method for preventing feature interaction problems. She points out that some interactions are desirable and that her method needs an analysis algorithm that generates a list of possible interactions among a set of features. Feature designers must adjust the feature specifications in an iterative process until the only remaining interactions are desirable ones.

How interactions can be detected depends on how features are specified and how interactions are defined. Bruns et al. [4] distinguish the following approaches:

- In the logical approach, features are specified as logical formulas and feature composition is logical conjunction. Feature interaction occurs if two features cannot be simultaneously satisfied.
- In the network specification approach [14], features are specified as sets of traces of network events and feature composition is set union. Feature interaction occurs if two feature sets intersect after certain operations are performed.
- In the operational approach [15], features are specified as processes and feature composition is some concurrent composition operation. Feature interaction occurs if the composed features fail to satisfy a global property such as deadlock freedom.
- In the feature as service transformer approach [4], a service describes the behaviour of a server that responds to input events. Feature composition is the successive transformation of a service by a sequence of features. Two notions of feature interaction are defined: two features interact (1) if the order in which they are applied affects the system behaviour, (2) if in some state, an input generates outputs that interfere.

Our work is a logical approach. What distinguishes it from other logical approaches, however, is the fact that we do not equate feature interaction with logical contradiction. In our opinion, logical contradiction is a sufficient but not a necessary condition for a feature interaction to appear. An example will be given in Section 7.

Other recent logical approaches are described in [13,16–18]. All of them use model checking techniques to detect interactions.

Jonsson et al. [16] propose a technique for hierarchically structuring requirements specifications in a way that simplifies change management and supports validation. As in our approach, requirements can be formulated and updated incrementally, supporting an evolutionary modelling of the application domain. Validation consists in checking iteratively the initial set of requirements (expressed in a linear time temporal logic) against the system model (expressed as a collection of automata), in scenarios where only a single feature is activated.

Like Jonsson et al., Felty and Namjoshi [17] use temporal logic to specify features and apply model checking to detect inconsistencies in the specification. In contrast to [16], they do not set up a separate system model, but convert feature specifications in ω-automata to perform the satisfiability test.

Khoumsi and Bevelo [18] have identified different kinds of interactions. Their interaction detection procedure is based on the search of special properties such as feature termination, variable consistency, events compatibility, event delayability and dependence on variables.

All of these detection procedures work on simplified models of the system. For example, telephone networks with only a fixed (and small) number of telephones are considered. The idea to detect interactions on simplified requirements or system models pertains also to our algorithm.

The work discussed so far is specifically designed to cope with the feature interaction problem. However, the notion of a *goal* [19] also provides a firm basis for detecting interactions between requirements. Van Lamsweerde and Letier use the concept of an *obstacle* or goal obstruction, which defines undesirable behaviour, to produce a refinement tree, the root of which is a goal negation. They define heuristics and formal techniques [20] to systematically generate obstacles from goal specifications and domain properties.

7 Discussion

We have presented an algorithm that helps to detect interactions in requirements. The algorithm is part of a more general method to systematically perform the first phases of software development. A systematic analysis of interactions leads to a better understanding of the requirements and avoids costly changes in later phases. The approach we have presented is domain independent and is method rather than language oriented.

It is useful not only for new systems but also for the evolution of systems. System evolution is motivated by changing requirements. Either new requirements are introduced or old requirements are replaced by different ones. In much the same way as for new systems, our algorithm can be used to analyse the consequences of changing the requirements before any changes to the software system are made.

We find it important that the detection of feature interactions be independent of the order in which the features are added, because this order may be arbitrary and insignificant. Moreover, we do not attempt to resolve feature interactions automatically. Such decisions are best taken by the customers.

We have already noted that it is important to find logical contradictions in requirements, but that not all interactions amount to logical contradictions. In the case study of an access control system [21], we had the following requirements: "when the door is unblocked, it will be re-blocked after 30 seconds" and "when a person has entered the building, the door will be re-blocked". These requirements interact, because it is intended to block the door immediately after the person has entered and not only after 30 seconds. Logically, however, the two requirements are not contradictory. It would suffice to re-block the door after 30 seconds, no matter if the person has entered or not. Hence, our algorithm can detect interactions that cannot be detected with logical procedures that detect only contradictions.

The approach for detecting feature interactions is truly heuristic. Its virtue lies in the fact that interactions on the requirements level can be detected very early, before a 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. Using our procedure, customers must inspect much fewer candidates than if a complete analysis, i.e. an inspection of all previously accepted constraints, 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, as is shown in [7,8].

Acknowledgement. We thank Thomas Santen for his comments on this paper.

References

1. M. Calder and E. Magill, editors. *Proc. 6th Feature Interaction Workshop, FIW 2000.* IOS Press Amsterdam, 2000.
2. K. Kimbler and W. Bouma, editors. *Proc. 5th Feature Interaction Workshop, FIW 1998.* IOS Press Amsterdam, 1998.
3. R. Turner, A. Fuggetta, L. Lavazza, and A. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999.
4. G.B. Bruns, P. Mataga, and I. Sutherland. Features as Service Transformers. In Kimbler and Bouma [2], pages 85–97.
5. P. Zave. Systematic design of call-coverage features. In *Proc. 7th International Conference on Algebraic Methodology and Software Technology*, LNCS 1548. Springer-Verlag, 1999.
6. M. Heisel and J. Souquières. A heuristic approach to detect feature interactions in requirements. In Kimbler and Bouma [2], pages 165–171.
7. 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.

8. M. Heisel and J. Souquières. De l'élicitation des besoins à la spécification formelle. *Technique et science informatiques*, 18(7):777–801, 1999.
9. 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.
10. P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.
11. M. Fowler and K. Scott. *UML distilled. Applying the standard Object Modelling Language.* Addison-Wesley, 1997.
12. M. Jackson and P. Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.
13. M. Plath and M. Ryan. Plug-and-Play features. In Kimbler and Bouma [2], pages 150–164.
14. A. Aho, S. Gallagher, N. Griffeth, C. Schell, and D. Swayne. Scf3/sculptor with Chisel: Requirements engineering for communication services. In Kimbler and Bouma [2], pages 45–63.
15. K.E. Cheng. Towards a Formal Model for Incremental Service Specification and Interaction Management Support. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interaction in Telecommunication.* IOS Press Amsterdam, 1994.
16. B. Jonsson, T. Margaria, G. Naeser, J. Nystrom, and B. Steffen. Incremental Requirement Specification for Evovlving Systems. In Calder and Magill [1], pages 145–162.
17. A. Felty and K. Namjoshi. Feature specification and automatic conflict detection. In Calder and Magill [1], pages 179–192.
18. A. Khoumsi and R.J. Bevelo. A detection method developed after a thorough study of the contest held in 1998. In Calder and Magill [1], pages 226–240.
19. A. van Lamsweerde and E. Letier. Integrating Obstacles in Goal-directed Requirements. In *Proc. of the 20 th International Conference on Software Engineering, ICSE'98*, Kyoto, Japan, 1998. IEEE.
20. A. van Lamsweerde and E. Letier. Handling obstacles in goal-directed requirements engineering. *IEEE Transactions on Software Engineering*, 2000. Special Issue on Exception Handling.
21. J. Souquières and M. Heisel. Une méthode pour l'élicitation des besoins: application au système de contrôle d'accès. In Yves Ledru, editor, *Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2000*, pages 36–50. LSR-IMAG, Grenoble, 2000. http://www-lsr.imag.fr/afadl/Programme/ProgrammeAFADL2000.html.
22. J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.

A Formal Expression of Constraints on Traces

In the following specification of system traces, we use the Z notation [22]. 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 fulfils the axioms of a partial ordering relation.

$[STATE, EVENT, TIME]$

TraceItem

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

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.

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

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

$Tr : \mathbb{P}\ TRACE$

$\forall tr : Tr;\ tr' : TRACE \mid$
$\quad tr'\ \mathrm{prefix}\ tr \bullet tr' \in Tr$

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.

$remove : TRACE \times \mathbb{P}\ EVENT \to TRACE$

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

B Formal Versions of Requirements and Facts

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

$\forall tr : Tr \bullet \forall i : \mathrm{dom}\ tr \bullet$
$\quad (tr(i).e = open \Rightarrow door\_closed(tr(i).s) \land$
$\quad (tr(i).e = close \Rightarrow door\_open(tr(i).s)$

Requirement 1. The lift is called by pressing a button.

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

Requirement 2. Pressing a call button is possible at any time.

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

Requirement 3. A call is served when the lift arrives at the corresponding floor.

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

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.

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

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

Requirement 5. When the lift has stopped, it will open the door.

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

Requirement 6. When the lift door has been opened, it will close automatically after d time units.

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

Requirement 7. The lift only changes its direction when there are no more calls in the current direction.

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

Requirement 8. When the lift is halted at a floor with the door open, a call for this floor is not taken into account.

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

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.

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

Requirement 10. Whenever the lift moves, its door must be closed.

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