A Heuristic Approach to Detect Feature Interactions in Requirements

Maritta Heisel, Jeanine Souquières

A Heuristic Approach to Detect Feature Interactions in Requirements

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 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 [HS98a]. 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. All system features must be reflected in these requirements, which are expressed formally as constraints on sequences of events that can happen or operations that can be invoked in the context of the system[1]. These constraints not only form the starting point for the development of the formal specification. They also support 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. 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. The steps of an agenda may have validation conditions associated with them that state necessary semantic conditions that the artifact must fulfill in order to serve its purpose properly.

2 Agenda for Requirements Elicitation

Our method for requirements elicitation is inspired by object oriented methods such as Fusion [CAB+94], and by the work of Jackson and Zave [JZ95]. It is performed in six steps. 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] A feature can correspond to one or several constraints.

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.
   We express requirements, assumptions, and facts by 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.

Using constraints to talk about the behavior of the system has the advantages that, first, it is possible to express *negative* requirements, i.e., to require that certain things do not happen. Second, it is possible to give scenarios, i.e., example behaviors of the system. Third, giving constraints does not fix the system behavior entirely. Constraints do not restrict the specification unnecessarily. Any specification that fulfills them is permitted.

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. It is performed in six steps. We illustrate the steps that are important for feature interaction detection by the case study of a lift system [HS98b].

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. Schematic expressions have the 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 than the postcondition. If the constraint is an invariant of the system state, we use the implication symbol $\Rightarrow$ instead of the symbol $\rightsquigarrow$. Transforming a constraint into its schematic form, we abstract from quantifiers and from parameters of predicate and event symbols.

   Example: A lift system could have the requirement "When the lift has stopped, it will open the door." The corresponding schematic expression is $stop \rightsquigarrow open$.

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 $pl \leftsquigarrow e$.
   Example: The door can only open when it is closed: $door\_closed \leftsquigarrow open$

   (b) Events establishing predicate literals.
   For each predicate literal pl, we need to know the events e that establish it: $e \rightsquigarrow pl$
   Example: The predicate *door_closed* is established by the event *close*: $close \rightsquigarrow door\_closed$

   (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: ${}_{\Rightarrow}p = \{pl : \neg\, p_{\Rightarrow} \bullet \neg\, pl\}$
   - the set of predicate literals that entail its negation: ${}_{\Rightarrow}\neg\, p = \{pl : p_{\Rightarrow} \bullet \neg\, pl\}$

   Note that only two of the four sets must be determined explicitely.
   Example: *door_closed* implies $\neg$ *door_open*: $door\_closed_{\Rightarrow} = \{\neg\, door\_open\}$

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 customers to decide if the conjunction of the new constraint with the candidates yields an unwanted behavior or not, and how detected interactions can be resolved.

6. If an interaction occurs, take one of the following actions: (i) correct a fact, (ii) relax a requirement (usually by adding a new pre- or postcondition, as preconditions are usually conjunctions, and postconditions are usually disjunctions), or (iii) strengthen an assumption.

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

4 Determining Interaction Candidates

In general, two constraints are interaction candidates for one another if they have common preconditions, but incompatible postconditions, as is illustrated in Figure 1. The left-hand side of the figure shows the situation where the incompatibility of postconditions manifests itself in the state immediately following the state that is referred to by the precondition. The right-hand side shows that the incompatibility may also occur at a later state.

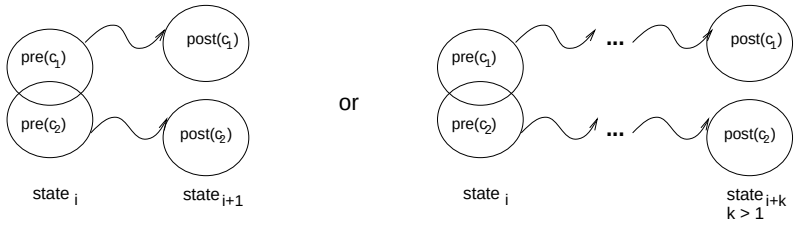

Figure 1: Interaction candidates

Our method to determine interaction candidates consists of two parts: *precondition interaction analysis* determines constraints with preconditions that are neither exclusive nor independent of each other. This means that there are situations where both constraints might apply. Their postconditions have to be checked for incompatibility. *Postcondition interaction analysis*, on the other hand, determines as candidates those constraints with incompatible postconditions. If in such a case the preconditions do not exclude each other, an interaction may occur.

4.1 Precondition Interaction Analysis

To decide if two constraints[2] $\underline{x} \rightsquigarrow \underline{y}$ 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 ($\underline{x} \cap \underline{u} \neq \emptyset$), 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 ($p \leadsto e$), then we also have detected a common precondition of the two constraints.

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

Figure 2 shows the general approach to find interaction candidates of the precondition for a new constraint among the facts, assumptions, and requirements already defined.

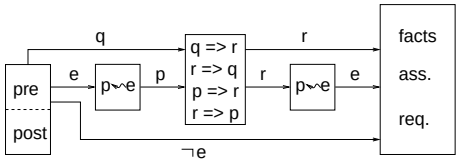

Figure 2: Candidates for precondition interaction

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 and constraint c, we define

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

With these preliminaries, we can define

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

[2] Underlined identifiers stand for sets of literals.

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 determined 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 usually it is impossible to infer anything from the non-occurrence of an event[3].

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 Analysis

To determine 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 3 gives an overview of the procedure.

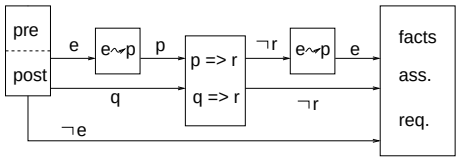

Figure 3: Candidates for postcondition interaction

We need the auxiliary definitions

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

where ls_1, ls_2 are sets of literals and $\neg\ \neg\ l = l$. Now, we can define

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

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

[3]Otherwise, we simply would set up a table with entries $pl \not\rightsquigarrow \neg\ e$ and treat negative event literals in the same way as postive ones.

4.3 Empirical Results

We have used this automatic procedure to determine interaction candidates for a lift system [HS98b]. It turned out that compared to a complete analysis about 70% of the analysis effort could be saved. However, the procedure did not find the right interaction candidates when the constraints made statements about system states that are not consecutive (as shown on the left-hand side of Figure 1), but where indefinitely many intermediate states are possible (as shown on the right-hand side of Figure 1). The reason was that our semantic tables (see Section 3) did not contain enough information to detect such interactions. The necessary information, however, can be added systematically.

Constraints can be assigned a *distance*, which characterizes the number of intermediate states that are possible between the pre- and post states related by the constraint. For each constraint with a distance greater than one, additional information is needed. It can be expressed as scenarios that show on the one hand how to proceed one step from the beginning state (to perform precondition interaction analysis) and on the other hand one step that leads to the final state (to perform postcondition interaction analysis). When such scenarios are added to the sets of constraints, our procedure finds as candidates all constraints where an interaction actually occurs.

More case studies must be performed to find out if this enhancement suffices to find all practically occurring feature interactions and if the percentage of analyses saved is stable for different application domains.

5 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 automatic procedure. 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 infeasible. 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 influence the overall behavior of the system and hence should be taken by the system designers or customers.

References

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

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

[HS98a] M. Heisel and J. Souquières. Methodological support for requirements elicitation and formal specification. In *Proc. 9th IWSSD*, pages 153–155. IEEE Computer Society, 1998.

[HS98b] M. Heisel and J. Souquières. Detecting feature interactions – a heuristic approach. In G. Saake and C. Türker, editors, *Proc. 1st FIREworks Workshop*, Preprint 10/98, pages 30–48, Fakultät für Informatik, 1998. Univ. Magdeburg. Available via http://wwwiti.cs.uni-magdeburg.de/iti_db/veroeffentlichungen/98/SaaTue98.html

[JZ95] M. Jackson and P. Zave. Deriving Specifications from requirements : an Example. In *Proc. ICSE'95*, pages 15–24. ACM Press, 1995.