

Toward an Evolutionary Software Technology

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

Extended Abstract

1 Motivation

Existing software engineering techniques usually treat the case where a *new* software system has to be built. All documents are developed from scratch, without any reference to existing documents. However, this situation is no longer realistic, because in more and more software projects, no new systems are constructed, but existing systems are evolved and adapted to new requirements. Hence, a task that becomes more and more important is to engineer *existing* software. Methods for an evolutionary software technology are still missing, even though object orientation and component-based software engineering enhance the possibility to re-use existing software.

This paper does not present finished results but explores some paths that lead to a software technology tailored for the evolution of existing systems.

2 Basic Principles

Every software product is made up of several documents, for example requirements documents, code, user manuals, etc. An important question for an evolutionary software technology is the choice of an appropriate basis for system evolution. Which documents should be the starting point of evolution strategies?

In the end, the evolution of a software system leads to changing its code. Therefore, one possible approach is to base software evolution strategies on the code. However, this approach is not advisable for the following reasons:

- The motivation for changing the existing system are additional or changed *requirements*. Hence, the requirements must be taken into account when evolving a system. A situation where the code is changed without any explicit reference to a requirements or specification document is unacceptable, even if it may common practice today.
- Before changing the system, the consequences of the change should be analyzed. It may be the case that new requirements interfere with old requirements. Such an analysis is highly non-trivial, even if it is performed on a high-level representation of the system. Trying to perform it on the code, which is the most low-level document representing the software system, would make the task even more difficult.
- The different documents that make up the software system, such as requirements, specification, and code, must be kept consistent. An evolution strategy that is based on code will almost certainly lead to neglecting the other documents. The result would be an undocumented and hence unmaintainable system.

We conclude that system evolution strategies should be based on *abstract* descriptions of software systems, i.e., requirements or specifications. Usually, these are informal documents expressed in natural language. To obtain semantically well-founded and automatable system evolution strategies, however, one should choose *formal* documents as a starting point. It follows that evolutionary software technology needs two phases:

1. Specification of existing systems
This phase establishes the prerequisites for a systematic evolution of the system.
2. Systematic system evolution
This phase deals with how to evolve a system in a systematic manner.

In the following sections, we sketch an approach how to tackle these two tasks.

3 Phase 1: Specification and Structuring of Existing Software

In his article “Software Aging” [Par93], Parnas describes how the structure of a software system is gradually destroyed by changes that are made when evolving or “maintaining” the system. For an evolutionary software technology, it is of utmost importance to preserve that structure when changes are made. This task is much easier when the structure, i.e., the software *architecture* [SG96], is made explicit.

We already mentioned that using the requirements and the code is indispensable for systematic software system evolution. With the architectural description, we have identified a third important document. This leads us to the idea to *construct different representations of the system and mappings between them*. On the one hand, we have the requirements of the system, which are its most abstract representation. On the other hand, there is the executable code, which is the most concrete representation of the system. In between the two, there are the specification and the architecture, as shown in Figure 1.

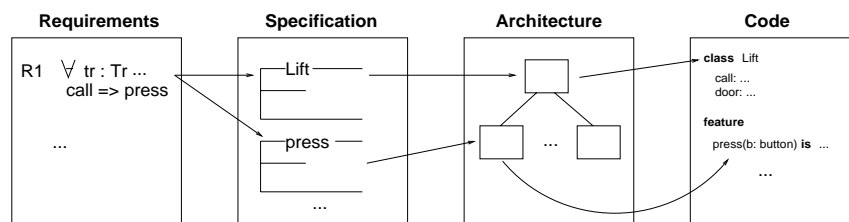


Figure 1: Representations and mappings

Having decided to use several different documents as the basis for system evolution, the tasks to be performed in the first preparatory phase of evolutionary software engineering consist in constructing different (formal) representations of the software system and mappings between these representations. The mappings, shown as arrows in Figure 1, constitute traceability links between the different parts of the various documents. For example, the mapping between the requirements and the specification shows for each requirement where it is reflected in the specification. Of course, the traceability links should be bi-directional. This means, for example, that it should not only be possible to find out how a requirement is distributed over the specification, but also to ask which requirements influenced the different parts of the specification.

Figure 1 just shows examples of possible intermediate representations of a system. One could also try to use fewer documents, for example do without the specification, or use more documents, for example the results yielded by reverse engineering tools as an additional representation between the architecture and the code [HK95]. The optimal number and nature of intermediate representations is still an open question. Too few intermediate representation result in very complex mappings, whereas too many documents result in an organizational overhead.

Another open question is *how* to construct the different representations and the mappings. A promising idea is to work from both ends, i.e., on the one hand from the requirements to the more concrete representations and on the other hand from the code to the more abstract representations. Both the requirements and the code should be available at the beginning of the first phase.

To validate the different representations and mappings, *consistency criteria* should be developed that help to detect errors early in the construction process.

If ever possible, the architectural description of the system should follow an *architectural style* [SG96]. Architectural styles characterize classes of systems that are structured according to the

same principles. Architectural styles and concrete architectural descriptions, although usually represented as informal diagrams, can be formalized. In contrast to informal diagrams, formal architectural descriptions have a precise meaning. This makes it possible to define criteria for a concrete architecture to belong to an architectural style [HL97] and to define operations on architectures that accommodate the changes that are necessary during system evolution.

In practice, a legacy system will hardly be an instance of an architectural style, and it will also have other flaws. Hence, the first phase will not only consist in constructing additional documents, but it will lead to a first revision of the system in order to make it amenable to systematic evolution.

The first phase of evolutionary software engineering, as sketched in this section, should not depend on the languages that are used to express the various documents. Instead, our goal is to develop a methodology that is representation independent.

4 Phase 2: System Evolution

Once a software system is represented in the way sketched in Section 3, its evolution can be performed in a systematic way. First, the new requirements must be expressed. They can either replace old requirements or be additional requirements that enhance the functionality of the system.

Next, the consequences of adding or replacing requirements should be analyzed. The new requirements could be incompatible with the already existing requirements. Such a situation is called an *interaction*. This term was originally coined in telecommunications and referred to different *features* a customer can subscribe to. A feature interaction occurs when combining different features leads to undesired or unexpected behavior or logical contradiction. Hence, we call the analysis of the consequences of adding new requirements to an existing system *interaction analysis*.

If interactions between requirements are detected, they should be resolved before proceeding with the system evolution. Resolution can either be achieved by changing (usually weakening) the new requirements, or by revising existing requirements. This process is an iterative one and must be repeated until no more interactions are found.

Once the set of requirements has stabilized, i.e., all interactions are resolved, the new requirements must be incorporated in the system. If existing requirements are replaced by similar ones, we can follow the mappings constructed in the first phase and change the intermediate documents one by one. The mappings show the places where changes must be made. In this case, the overall structure of the system is not likely to change.

The situation is more complicated if the system functionality is enhanced by adding new requirements. Then, the mappings between the different documents no longer indicate the places where changes have to be made, and the present architecture of the system might no longer be appropriate. Methods are needed to

- exploit the mappings as far as possible also for new requirements.
A possible approach is to classify the requirements, for example requirements that have to do with the user interface, or the update of data, etc. If a new requirement belongs to a class of requirements already present in the system, one could try to use the mappings belonging to that class.
- incorporate entirely new requirements into the system that have no similarity with existing requirements.
It seems that this activity has something in common with the first phase of evolutionary software engineering. We must work from an updated set of requirements and incorporate new requirements into all of the following more concrete documents.
- change a software architecture in a systematic way.
The change can either lead to a different architecture adhering to the same architectural style, or even entail a change of the architectural style of the system. To change software architectures, operators that work on architectural descriptions should be developed. These operators should also help to change the code.

So far, we have presented a general approach to evolutionary software engineering and have pointed out concrete research questions suggested by that approach. In the rest of the paper, we present a piece of work that is more mature than what was discussed before, and that makes up an important part of evolutionary software technology. That work is a heuristic algorithm to detect interactions in requirements. Such an algorithm is necessary for systematic software evolution, because the consequences of a system change must be analyzed before actually executing it.

4.1 Analyzing Requirements for Interactions

Given a set of already accepted requirements and a new requirement, the algorithm we present in the following 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. A heuristic algorithm is appropriate, because the notion of interaction can hardly be formalized. It covers more phenomena than just logical inconsistency¹. Striving for a provably correct and complete algorithm would necessitate a formal and decidable notion of interaction. However, it is questionable if such a definition is possible or even desirable.

The algorithm determines a set of candidates to examine. It does not prove that there really is an interaction between the new constraint and each candidate. It is up to the stakeholders of the system to decide if the combination of the new requirement with the candidates yields an unwanted behavior or if it even is contradictory.

This algorithm was not developed specifically for software evolution, but as part of a requirements engineering method [HS99]. For analyzing interactions, however, it does not make any difference whether some of the requirements are already implemented or not. Hence, the algorithm is just as useful for the evolution of systems as it is for new systems.

System view.

We take the following view of a system: the system is started in some state S_1 . When event e_1 happens at time t_1 , then the system enters state S_2 , and so forth:

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

An event can either come from the environment of the software system and be detected via sensors, or it can be the call of a system operation by a user. Hence, this view of a system is valid for both reactive and transformational systems.

Formalization of requirements.

Our requirements engineering method proposes to express requirements as constraints over the set Tr of admissible system traces, using *event* and *predicate* symbols.

We recommend to express – if possible – constraints as implications, where either the precondition of the implication refers to an earlier state or an earlier point in time than the postcondition, or both the pre- and postcondition refer to the same state, i.e. we have an invariant of the system.

Example. We consider an elevator. A possible requirement is: “When the lift passes by floor k , and there is a call from this floor, then the lift will stop at floor k ”.

$$\forall tr : Tr; k : Floor \bullet \forall i : \text{dom } tr \mid i \neq \#tr \bullet \\ \text{passes_by}(tr(i).s, k) \wedge \text{call}(tr(i).s, k) \Rightarrow tr(i+1).e = \text{stop}(k))$$

The symbols *passes_by* and *call* are predicate symbols, whereas the symbol *stop* is an event symbol. For each trace tr and each i -th element $tr(i)$ of the trace which is not the last one ($i \neq \#tr$), we require that if the predicates *passes_by* and *call* are true of the state $tr(i).s$ and the floor k , then the next event $tr(i+1).e$ will be *stop*.

Schematic expressions.

The algorithm to determine interaction candidates uses schematic versions of formalized constraints. These schematic expressions have the following form:

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

where the x_i, y_j are literals (i.e., either predicate or event symbols or their negations) and each \diamond denotes conjunction or disjunction. The symbol \rightsquigarrow separates the precondition from the postcondition.

¹Example: In the case study of an access control system [SH00b], 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.

For transforming a constraint into its schematic form, we abstract from quantifiers and from parameters of predicate and event symbols. This results in a (deliberate) loss of information. For example, $x \wedge \neg x$ are no longer contradictory, because x could refer to a different argument than $\neg x$.

Note that a detailed formalization of a requirement is not strictly necessary to set up the schematic expressions. The schematic constraint could also be obtained directly from the natural-language requirement.

Example. The above requirement has the schematic form $passes_by \wedge call \rightsquigarrow stop$

Semantic relations.

Because the set of interaction candidates is determined completely automatically, the algorithm cannot be based on syntax alone. We also must take into account the semantic relations between the different symbols. To that end, 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 $pl \rightsquigarrow e$.

Example. The event $stop$ can only occur if the elevator is not halted: $\neg halted \rightsquigarrow stop$

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 $halted$ is established by the event $stop$: $stop \rightsquigarrow halted$

3. Relations between predicate literals. For each predicate symbol p , we determine:

- the set of predicate literals it entails: $p \Rightarrow = \{q : PLit \mid p \Rightarrow q\}$

Example. $halted \Rightarrow = \{at, \neg passes_by\}$

- the set of predicate literals its negation entails: $\neg p \Rightarrow = \{q : PLit \mid \neg p \Rightarrow q\}$

Example. $\neg halted \Rightarrow = \{passes_by, \neg at\}$

Determining interaction candidates.

Two constraints are interaction candidates for one another if they have overlapping preconditions but incompatible postconditions, as is illustrated in Figure 2. “Incompatible” does not necessarily mean “logically inconsistent”; it could also mean “inadequate” for the purpose of the system.

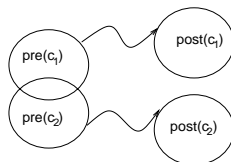


Figure 2: Interaction candidates

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.

Precondition interaction candidates. If two constraints² $\underline{x} \rightsquigarrow \underline{y}$ and $\underline{u} \rightsquigarrow \underline{w}$ 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 pl , and e is only possible if pl holds ($pl \rightsquigarrow e$), 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 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.

²Underlined identifiers denote sets of literals.

Figure 3 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.

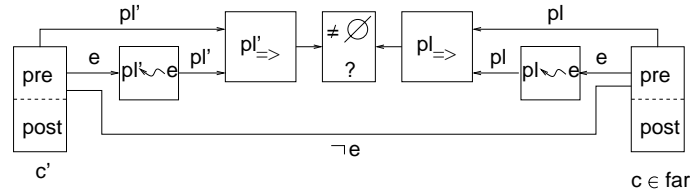


Figure 3: Determining interaction candidates by precondition analysis

Postcondition interaction candidates. To find conflicting postconditions, we compute the set of predicate literals that are entailed by the postcondition of the constraints under consideration. For an event e contained in the postcondition of a constraint, all predicate literals pl with $e \rightsquigarrow pl$ must be considered, too. If the postcondition of the new constraint entails a predicate literal whose negation is entailed by the postcondition of an already accepted constraint, these constraints are interaction candidates for each other. Figure 4 illustrates the definition.

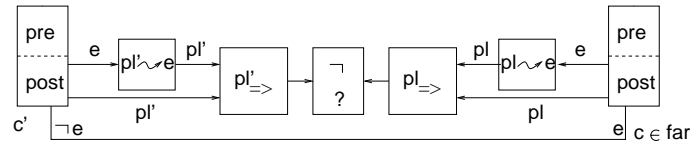


Figure 4: Determining interaction candidates by postcondition analysis

The algorithm is explained in more detail in [HS98b]. There, also the formal definitions of the candidate sets are given.

Example. To briefly illustrate the algorithm, we consider a new requirement: “The lift gives priority to calls from the executive landing”, whose schematic form is

$$call \rightsquigarrow next_stop_at_executive_floor$$

This requirement interacts with the one previously given: if there is a call from the executive floor, the elevator will not necessarily stop at the floor it currently passes by, even if there is a call for that floor. The algorithm correctly identifies the previously given requirement as an interaction candidate for the new one via the common precondition $call$.

The algorithm has been validated in several case studies. Besides the elevator [HS98a], we have treated a microwave oven, an automatic teller machine, a simple telephone system, an access control system [SH00b] and a light control system [SH00a].

5 Conclusions

In this paper, we have presented a general approach to evolutionary software engineering. This approach must be refined, and the research questions raised need further investigation. In summary, we consider the following points as important:

- Mastering systematic software evolution becomes more and more important.
- Software evolutions strategies should not be based on code, but on more abstract representations of the software system.
- These abstract representations will usually not exist for legacy systems. Hence, the first task to perform before systematic software evolution is possible is to construct these representations.
- Several representations of a system on different levels of abstractions seem to be useful. Bi-directional traceability links must be established between these representations.

- Before a system is changed, it should be analyzed if the new requirements do not interact in an undesirable way with the rest of the system. We have presented a heuristic algorithm to support this task.
- Once the new requirements have stabilized, the system can be changed in a systematic way, making use of the mappings constructed in the first phase. This task is more difficult when the new requirements make a re-structuring of the system necessary.
- Re-structuring a system should be supported by architectural operators.
- In connection with the mapping from the architectural description to the code, the architectural operators should help in changing the code according to the architectural changes.

References

- [HK95] Maritta Heisel and Balachander Krishnamurthy. Bi-directional approach to modeling architectures. Technical Report 95-31, Technical University of Berlin, 1995.
- [HL97] Maritta Heisel and Nicole Lévy. Using LOTOS patterns to characterize architectural styles. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, LNCS 1214, pages 818–832. Springer-Verlag, 1997.
- [HS98a] Maritta Heisel and Jeanine Souquière. Detecting feature interactions – a heuristic approach. In G. Saake and Can Türker, editors, *Proc. of the first FIREworks Workshop*, Preprint 10/98, pages 30–48, Fakultät für Informatik, 1998. Univ. Magdeburg.
- [HS98b] Maritta Heisel and Jeanine Souquière. A heuristic approach to detect feature interactions in requirements. In K. Kimbler and W. Bouma, editors, *Proc. 5th Feature Interaction Workshop*, pages 165–171. IOS Press Amsterdam, 1998.
- [HS99] Maritta Heisel and Jeanine Souquière. A method for requirements elicitation and formal specification. In Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, and Elisabeth Métais, editors, *Proceedings 18th International Conference on Conceptual Modeling, ER'99*, LNCS 1728, pages 309–324. Springer-Verlag, 1999.
- [Par93] D. L. Parnas. Software Aging. In *Proceedings International Conference on Software Engineering*. ACM Press, 1993.
- [SG96] Mary Shaw and David Garlan. *Software Architecture*. IEEE Computer Society Press, Los Alamitos, 1996.
- [SH00a] Jeanine Souquière and Maritta Heisel. A method for systematic requirements elicitation: Application to the light control system. Technical Report A00-R-090, LORIA, Nancy, France, 2000.
- [SH00b] Jeanine Souquière and Maritta Heisel. Une méthode pour l'élicitation des besoins: application au système de contrôle d'accès. In Yves Ledru, editor, *Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2000*, pages 36–50. LSR-IMAG, Grenoble, 2000. <http://www-lsr.imag.fr/afadl/Programme/ProgrammeAFADL2000.html>.