On the Evolution of Component-based Software

Isabelle Coté, Maritta Heisel, Jeanine Souquières

On the Evolution of Component-based Software

Isabelle Côté[1], Maritta Heisel[1], and Jeanine Souquières[2]

[1] University Duisburg-Essen, Faculty of Engineering, Department of Computer Science and Cognitive Science, Workgroup Software Engineering, Germany
[2] LORIA – Nancy Université
B.P.239 Bâtiment LORIA
F-54506 Vandœuvre-les-Nancy

Abstract. In a component-based software system the different components communicate with each other through their interfaces, possibly using adapters. Whenever the requirements or the environment change, the component-based software system must be adapted accordingly. We present a method that supports a systematic evolution of component-based software at the architectural level. It is based on operators and modification schemata that help to identify different architectural alternatives to realize the evolution task.

1 Introduction

The component-based paradigm has received considerable attention in the software development field in academia as well as in the industry in the past years. A component-based application consists of assembled, pre-fabricated components. Some of these components are considered to be black-boxes, i.e., their implementation cannot be changed as it is inaccessible. They possess interfaces that describe their visible behavior. In general, interfaces are not directly compatible, and an adapter is required to bridge this gap.

Assembling a software system in a component-based way leads to well-structured and modular architectures with independent and re-usable components. The component-based paradigm provides a challenge for software evolution, i.e., the task of adapting the software system to new or changed requirements and/or environment. This is because the components themselves usually cannot be manipulated. It is only possible to rely on their interfaces for this matter. This, in turn, has an influence on the solution of the evolution task, as not all possible solutions are implementable. Therefore, an evolution method for component-based systems has to be tailored to address these special conditions. With this work, we propose a method that allows for a systematic evolution of component-based systems.

We focus on the architecture of the system to be evolved, relying on available documents. We propose different architectural evolution schemata that guide the engineer to determine different alternatives for the evolution to be performed. The method then provides some heuristics to help choosing from the different alternatives. Our method is intended to be carried out by human developers, even though some steps may be tool-supported.

The paper is organized as follows. In Section 2, we introduce the basic terminology used as well as the prerequisites under which our method can be applied. An overview of our evolution method is presented in Section 3, introducing the different schemata. In Section 4 we introduce the access control system, which serves as a case study and forms the starting point for our evolution. We then illustrate the application of the proposed method by evolving the access control system. Related work is discussed in Section 5. A summary and future work conclude the paper.

2 Basic terminology and Prerequisites

According to Jackson and Zave [1], the task in software engineering is to build a *machine* that operates in an *environment* and serves to improve that environment. The overall purpose the machine should serve is called the *system mission*. *Requirements* are *optative* statements expressing our wishes how the environment should behave, once the machine is in operation. Accordingly, requirements do not refer to the machine but only to the environment; they refine the system mission. To build the machine, a *specification* has to be constructed. Specifications are *implementable* requirements. They describe the machine and are the starting point for its construction. To transform requirements into specifications, *domain knowledge* is used. It consists of *indicative* statements, expressing facts and assumptions that are true in the environment.

For example, a requirement for an access control system is that only those persons are admitted to enter a building who have permission to be in the building. Such a requirement is not implementable because a software system does not know who has permission. To transform this requirement into a specification, we use the knowledge that each person who wants to enter the building possesses an entry card containing a user identification. Furthermore, a database is used to store which user has permission for the building. With this domain knowledge, we can derive the following specification: *"Only those persons with an entry card are admitted to the building for whom the database contains an entry specifying that they have permission to be in the building."* This optative statement is now implementable.

In this paper, we investigate how *component-based* software can be evolved in a systematic way. Thus, we consider the case where a machine, which is built from *components*, is already developed. A software component is a piece of software that is encapsulated and accessible only via well-defined interfaces. Components can be integrated into different environments. In general, it is not possible to access (and therefore, change) their implementation. To adapt a software component to different environments, *adapters* are used to connect different components whose interfaces do not coincide completely.

Our evolution method relies on several documents that need to be available. These are either available from the start or need to be reconstructed. Even in the first case it is advisable to consolidate the documents to make sure that they are up-to-date. First, prior to the evolution, the current situation should be described. Usually, this is done by writing a short text, providing an as-is status of the system and the shortcomings which have been identified. The shortcomings are the reason for the evolution task. Then, *evolution requirements* (eRs) are derived, based on the shortcomings stated in the

descriptive text. We assume that the eRs are available, have been consolidated and are non-contradictory.

Second, the *machine specification* should be known, i.e., its behavior at its external interfaces. We use UML 2.0 sequence diagrams [2] to express these specifications. An example can be found in Fig. 7.

Third, we need the component-based software architecture of the machine to be evolved. It can be represented, for example, by UML composite structure diagrams, see Fig. 3. Such diagrams contain named rectangles, called *parts*. They represent the components the software is built from. Parts may have *ports*, denoted by small rectangles. Ports may have interfaces associated to them. Provided interfaces are denoted using the "lollipop" notation, and required interfaces using the "socket" notation. The interfaces are described using interface classes known from UML. We distinguish different component layers in the architecture: one or more components implement those parts which are needed to fulfill the system mission (called *application* components). These application components are clearly separated from components handling auxiliary functionality (called *adapters*). An application component as well as the adapters may be modified when performing an evolution, whereas the third-party (black-box) components cannot be changed.

Finally, the communication between the different components must be specified, using again sequence diagrams. Figure 5(a) shows an example of the specification of such communication. Note that these specifications should always have a *precondition* expressing in which state the interaction begins, and a *postcondition* expressing in which state the interaction ends. The pre- and postconditions are denoted as state invariants in the sequence diagrams.

3 The Evolution Method

3.1 Expressing Evolution Tasks

We have identified two basic constituents, namely *operations* such as *add*, *modify*, *replace*, and *delete* on the one hand and *elements* such as *requirement*, *environment*, and *component* on the other hand. Based on these two constituents we form pairs that specify the kind of change that is required and which element is affected by the given evolution task. The basic operations have the following meaning:

- *add*: something that is not yet present in the software system and which should be newly introduced, e.g., a light for users at an entry turnstile.
- *modify*: something that already exists, but has to undergo some modification in order to cope with a changed situation, e.g., a visual signalization is used differently than before.
- *replace*: a present part in the system should be exchanged by a new part, e.g., the light bulb of a traffic light is replaced by a LED bulb with its corresponding software driver.

  In general, the behavior of the software system should not change when performing a replacement. It may, however, be necessary to adapt the application component or the adapters.

Algorithm 1 Evolution method in pseudo code notation

module evMethod(**in** *req_set*:Set(eRs),**inout** *sw*:Software, *docs*:Documents)

```
1: investigate(in sw, req_set, inout docs, out classi_set)
2: deriveExtSpec(in classi_set, inout docs, out ev_set)
3: while ev_set ≠ ∅ do
4:    selectEspec(in ev_set, out curr_spec)
5:    deriveAltern(in curr_spec, inout docs, out alt_set)
6:    assessAltern(in alt_set, docs, out as_set)
7:    selectcand(in as_set, out cand)
8:    deriveIntSpec(in cand, inout docs, out spec)
9: end while
10: evolve(in spec, inout sw, docs)
```

end module

– *delete*: something which has become superfluous is removed, e.g. an exit turnstile may no longer be needed. This case will not be considered in the rest of the paper.

The basic elements have the following meaning:

– *requirement*: statements expressing our wishes how the environment should behave, once the machine is in operation.
– *environment*: that part of the "real world" which is relevant to our problem.
– *component*: the black-box entities we possess.

However, not all combinations make sense. The prohibited combinations are:
Add, delete component: Usually, adding or deleting a component becomes necessary by some reason which lies in adding, deleting or modifying requirements and/or the environment. A component itself does not account for its addition or deletion.

Modify component: Components are considered as black-boxes and their source code is not supposed to be accessible.

Replace requirement: We treat a replacement as a special kind of modifying a requirement.

Note that additions, modifications, or deletions in the environment usually influence at least one requirement. Therefore, applying an operator involving the environment usually triggers the application of another operator involving a requirement. Hence, we consider environment and requirement as one entity. However, evolving requirements need not imply effects on the environment.

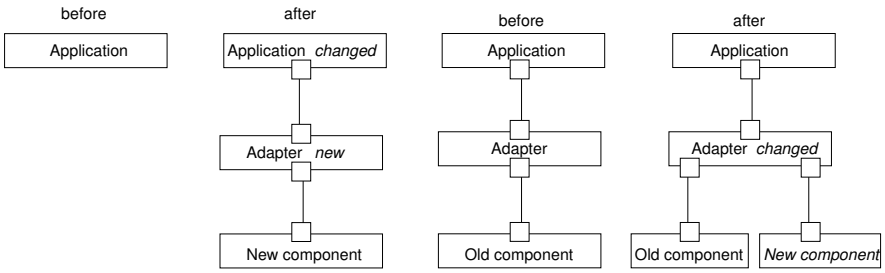
Schemata We have developed *architectural evolution schemata* that specify how a component-based software architecture can evolve in different cases. Each schema specifies which parts of the software are affected by the evolution and which parts will remain unaffected. These schemata will be detailed at the appropriate places in Subsect. 3.2.

3.2 The Evolution Method

To describe our method we use a pseudo code-like notation containing procedure definitions and procedure calls (cf. Alg. 1). The procedure parameters are characterized

as follows: **in** These parameters constitute input parameters, which means that they are read but not changed. **inout** The parameters here are read and may be changed, as well. **out** These parameters are output parameters. They are generated by the procedure. In the following, we detail the different steps of Alg. 1:

investigate (line 1) The first step is to get an *understanding* of the software (**in** parameter *sw*) and its components. The actions to be performed are to verify whether all relevant documents (**inout** parameter *docs*) are available and up-to-date. Should this be not the case, it is necessary to reconstruct the missing documents. After this, every eR is classified according to the combinations described in Subsect. 3.1. Furthermore, it is necessary to decide if the eR is mission-critical, i.e., it is needed to fulfill the system mission. Here, the following question provides some guidance on finding a solution: *"Is the system still capable of fulfilling its purpose even if the new functionality fails?"* According to the answer to this question the eRs are being tagged as either mission-critical (answer: no) or not mission-critical (answer: yes). The classified and tagged eR then form the set of classified evolution requirements *cRs* (**out** parameter *classi_set*).



(a) Schema I (b) Schema II

Fig. 1. Schemata for Addition.

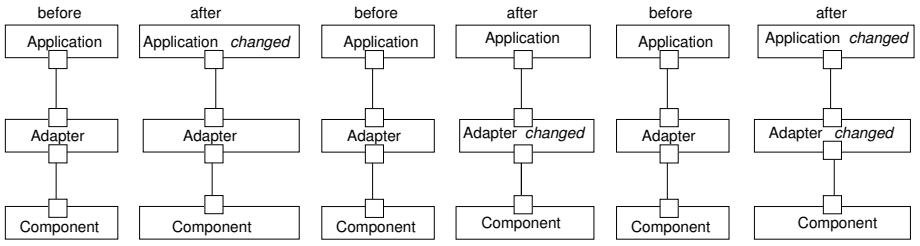
deriveExtSpec (line 2) As a second step, we *derive a specification* for every cR (**in** parameter *classi_set*). The specifications we consider here are *machine specifications* i.e. they describe the machine behavior that is visible at the external interfaces. The following actions have to be performed for the different classifications: *add(requirement, environment), modify(requirement, environment)*: transform the evolution requirement into a machine specification; update the domain knowledge if necessary. *Replace(environment), replace(component)*: update the domain knowledge. The obtained specifications form the set of machine specifications (**out** parameter *ev_set*).

select**Espec (line 4) As a next step, we select one of the machine specifications. How the selection is performed depends on the given circumstances.

deriveAltern (line 5) For the selected specification (**in** parameter *curr_spec*) we derive possible alternatives solving the problem. For that purpose, we need to ask ourselves:

"How can this behavior be realized internally?" This is necessary, because at the moment we only have a machine specification describing the external behavior that is visible to the environment. In particular, we have to inspect all pre- and postconditions that are used in the available internal specifications. Then, we have to decide if we find conditions that can serve as a pre- or a postcondition for the new specification.

The actions to be performed for the different evolution cases are:



(a) Schema II (b) Schema III

Fig. 2. Schemata for Modification.

– *add(requirement, environment)*
  Collect all sequence diagrams related to the evolution task. Compare the resulting diagrams according to either common starting states (preconditions) or common terminating states (postconditions) with the selected evolution specification. This results in the following cases:

  - *Schema I for Addition* (Fig. 1(a)). The left-hand side shows the current state with an application component which may or may not have interfaces to other connected components (not shown in the figure). This schema adds a new interface to the application which connects it to the new component.
    If no common state can be found in the sequence diagrams, this schema should be applied. Note that it is always applicable.
  - *Schema II for Addition* (Fig. 1(b)). In contrast to the previous schema, it is not necessary to add a new interface to the application component. Instead, the new component is handled together with an already existing component. The corresponding adapter is changed to mediate the communication of the two (or more) components with the application. The already existing interfaces remain unchanged and a new interface has to be added to the adapter.

    If an internal specification can be found that shares a state with the evolution specification, then this schema can be applied; depending on the following cases:

    * Both specifications share the same precondition: the application informs the adapter about the met precondition. The adapter then sends appropriate signals to the old as well as the new component.

    - Both specifications share the same postcondition: the old and the new component inform the adapter that they established the terminating condition. The adapter then processes this information and informs the application.
    - The precondition of the existing internal specification is the postcondition of the evolution specification: the new component informs the adapter that it established the postcondition; the adapter then informs the old component that the precondition is met.
    - The postcondition of the existing internal specification coincides with the precondition of the evolution specification: analogues to the previous case.

– *modify(requirement, environment)*
  We have both the original specification as well as the modified and new additional sequence diagrams, respectively. The different schemata for modifying are:
  - *Schema I for Modification* (Fig. 3.2). The original and modified specification are sufficiently similar. The modification can be handled completely by the corresponding application component.
  - *Schema II for Modification* (Fig. 2(a)). The modifications in the specification are related to the adapter.
  - *Schema III for Modification* (Fig. 2(b)). The modified specification relies on information that was not processed in the current version, e.g., it uses services the component provides but have not been of interest previously. Then, the application as well as the adapter have to be modified.

– *replace(environment)*, *replace(component)*
  It is necessary to investigate the interfaces which are already present as well as the interfaces of the new component:
  - *Schema I for Replacement* (Fig. 4(a)). This schema can be applied if the new component is sufficiently similar to the old one and the data that is required or provided by this new component can be converted by the adapter. The application as well as the interfaces between adapter and application remain unaffected.
  - *Schema II for Replacement* (Fig. 4(b)). The interfaces between the application and the adapter cannot be retained. This is usually the case when the data being transmitted changes and the conversion cannot be performed by the adapter alone.

Note that when deriving the different alternatives, it may become apparent that further evolution tasks are needed in order to fulfill the current evolution task. Whenever this occurs, the method has to be applied for every resulting additional evolution requirement.

assessAltern (line 6) Subsequently, it is necessary to decide which of the determined alternatives should be chosen. Here we rely on the tags introduced during the investigation step (cf. Alg. 1 line 1): If the task is tagged with “not mission-critical”, schemata leaving the application component unaffected are to be preferred. The rule of thumb is: *“The application should only be changed if the new functionality is mission-critical.”*

All the alternatives determined in *deriveAltern* constitute valid solutions. However, each alternative with its underlying schema has its advantages and disadvantages. These

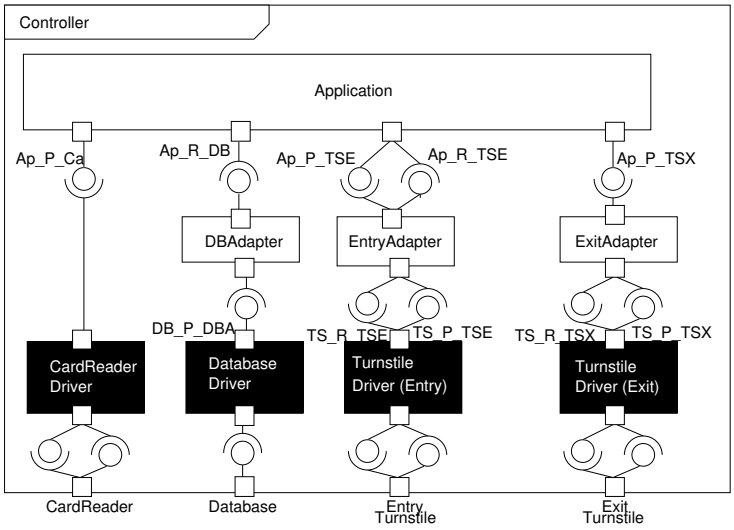


Fig. 3. Software architecture of the access control system.

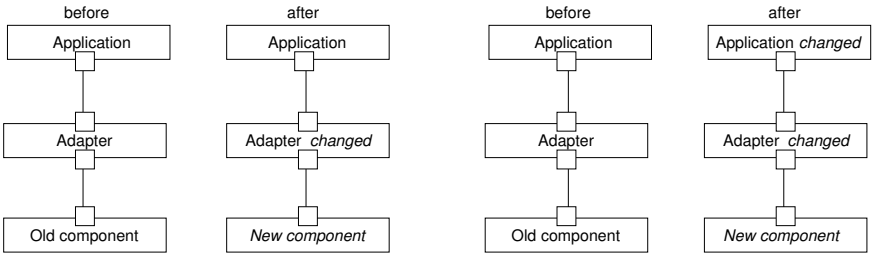


Fig. 4. Schemata for Replacement.

have to be weighed in order to find the schema which fits best under the given circumstances:

- *Schema I for Addition* provides the advantage of guaranteeing a well-structured architecture. Each component remains independent and reusable. On the other hand the application might get too complex, which would make future evolutions difficult.
- *Schema II for Addition* does not require changes of the application component. The possibility to arrange the components in several ways (grouping old and new components together), based on the same schema is an advantage. However, some solutions may complicate further evolution tasks.

  Furthermore, the adapter may take over too much functionality, mutating into an application-like component. It should be avoided that the adapter gets too complex.

- *Schema I for Modification* makes changes to the application, which seems to be a drawback. However, as only the application needs to be changed (all the other parts remain unaffected), this also constitutes the advantage of this schema.
- In contrast to the previous schema, *Schema II for Modification* only requires changes to the adapter in order to implement the modified functionality. Therefore, this schema should be preferred whenever possible.
- *Schema III for Modification* requires changes to the adapter as well as to the application. If it is possible, one of the other modification schemata should be preferred.
- *Schema I for Replacement* constitutes the desirable case when replacing a component as it only requires changes to the adapter. Similar to *Schema II for Addition*, it has to be avoided that the adapter gets too complex.
- *Schema II for Replacement* should be avoided where possible if the new functionality is not mission-critical, as many changes are necessary in order to implement the replacement.

These advantages and disadvantages should be taken into consideration when choosing between different evolution alternatives. They can serve as heuristics or rules-of-thumb. However, for the final decision human comprehension and experience are necessary.

***select**$_{cand}$* (line 7) After assessing the alternatives (**in** parameter *as_set*), we *select* the alternative (**out** parameter *cand*) that is best suited for the current problem based on the reasoning performed in step *assessAltern*.

deriveIntSpec (line 8) For this alternative (**in** parameter *cand*) we derive the internal specification (**out** parameter *spec*).

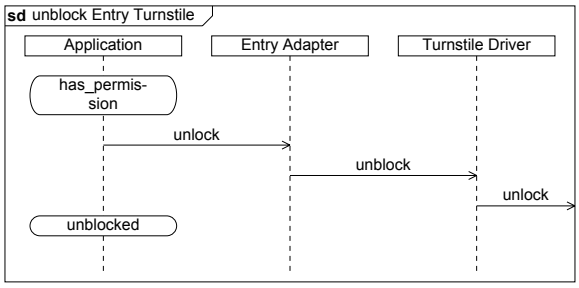
evolve (line 10) We now incorporate the chosen alternative into the software system.

4 Application on the Case Study

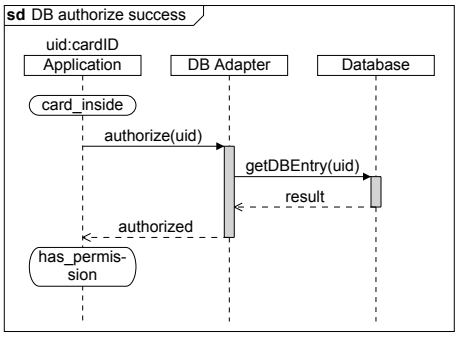
4.1 The Base System

We evolve a simple access control system already used for other investigations [3], which controls the access to a building. Persons who are authorized to enter the building are equipped with a smartcard on which a user identification is stored. The access control system queries a database to obtain the information whether the person is permitted to enter the building. If access is granted, a turnstile located at the entrance is unblocked, so that the person can enter the building. At the exit of the building, another turnstile is installed. It is always unblocked and only serves to count the number of persons who have left the building.

The software architecture for the controller is presented in Fig. 3. The top-most layer consists of the application component, which implements the mission-critical functionality. The bottom-most layer includes the software drivers. These drivers connect the software to the hardware components and cannot be changed. Therefore, they are shown in black. The middle layer consists of adapters. The adapters are used to connect the application component to the software drivers. In some cases it may be possible to omit



(a) Unblocking Entry Turnstile



(b) DB authorize

Fig. 5. Internal specifications.

such an adapter. The connection between the different interfaces is realized through *required* and *provided* interfaces (cf. Fig. 3). Thus, adapters implement the required services by the provided ones.

The required and provided interfaces in our software architecture are named according to the following convention, consisting of a compound of abbreviations. The first abbreviation, usually denoted by two to three letters of the component, indicates the first component e.g. *Ap* for application. The second abbreviation is either "P" for provided or "R" for required. The third abbreviation, indicates the second component to which the first component is connected via the interface, e.g. *TSE* for the entry turnstile adapter. Hence, the name *Ap_P_TSE* describes the provided interface of the application connected to the entry adapter.

4.2 Evolution: Visual Signalization

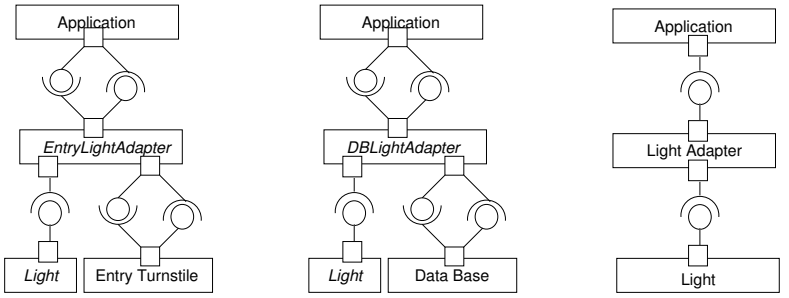
The original system does not possess any kind of visual signalization. Therefore, we can formulate the following evolution requirement (eR):
(eR1) *Add a green light that is lit for* n *seconds when a person is authorized to enter the building.*

investigate (line 1) The first step now is to *investigate* the software. In our case, we assume that all prerequisites stated in Section 2 are fulfilled. Taking a look at our eR, it can clearly be classified as *add(requirement, environment)*. Hence, we get $\mathrm{cR1}_{add(r,e)}$. As the second step of the classification, we evaluate whether it is mission-critical. This can clearly be answered with "no". Thus, we tag $\mathrm{cR1}_{add(r,e)}$ as not mission-critical, resulting in $\mathrm{cR1}^{-}_{add(r,e)}$, which is added to the set of classified evolution requirements.

deriveExtSpec (line 2) For every member of the cRs (parameter *classi_set*) we derive a machine specification. The corresponding machine specification for cR1 can be stated as follows, applying the rules for *add(requirement, environment)*:

(eS1): If *has_permission*[3] , turn on green light for *n* seconds.

The sequence diagram of Fig. 7 illustrates this specification. We can then add it to the evolution set (parameter *ev_set*).



(a) same precondition (b) postcondition matches precondition (c) handling *Light* separately

Fig. 6. Different solutions for Scenario I.

select**Espec (line 4) As we only have one member in our set, we select it.

deriveAltern (line 5) With the current specification we derive all possible alternatives. For our example, we are able to find two sequence diagrams in the initial system, namely *unblock Entry Turnstile* (cf. Fig. 5(a)) and *DB authorize* (cf. Fig. 5(b)). They can be mapped to the following alternatives:

– *Alternative I.* We group properties together according to a common starting point, i.e., their common precondition (cf. *has_permission* in Figs. 7 and 5(a)). The idea behind this alternative is that it signalizes that a person is authorized at the same time the turnstile is unblocked. This corresponds to applying *Schema II for Addition*.

[3] *has_permission* is the state reached when the database component sends the message *authorized = true*

Figure 6(a) illustrates the corresponding architecture according to the application of this schema. A new component *Light* is connected to the adapter of the *Entry Turnstile*. The *Entry Adapter* is renamed to *EntryLightAdapter*. Originally, the adapter translated the signal *unblock* of the *Application* in the signal *unlock* provided by the *Entry Turnstile*. The evolved adapter still sends the message *unlock* to the *Entry Turnstile*. In addition, it sends the signal *turn_on* to the *Light* via the newly introduced interface, as presented in Fig. 8.

- *Alternative II*. The sequence diagram shown in Fig. 5(b) specifies the communication between the *Application* and the *Data Base*. Its postcondition *has_permission* matches the precondition of the sequence diagram *Light behavior* (cf. Fig. 7).
  In this case, we also apply *Schema II for Addition*. The resulting architecture shown in Fig. 6(b) looks similar to the one in *Alternative I*: the adapter is renamed into *DBLightAdapter*, and a new interface it added, as well. However, the change performed in this case is different. The adapter informs the *Application* that the person is authorized to enter the building via the message *authorize(uid)*, which establishes the postcondition *has_permission(uid)*. Then, the adapter sends the message *turn_on* to the *Light*, as presented in Fig. 9(a).
  Note that in both cases the adapters must have access to a timer component in order to deal with the timing constraints, i.e. to handle the necessary timeouts.
- *Alternative III*. It introduces an independent component *Light*. In this case, we do not have to take into account any existing sequence diagrams.
  However, it is necessary to add a new port to the application component with corresponding interfaces (cf. Fig. 6(c)). The application knows that the precondition is met and sends the signal *turn_on* to the light driver (cf. Fig. 9(b)).

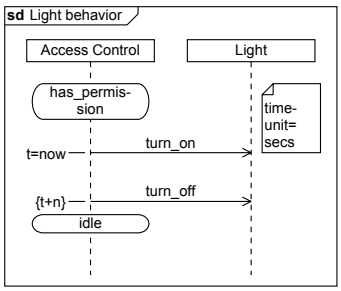


Fig. 7. Machine specification eS1.

With this, we have our set of alternatives (parameter *alt_set*).

assessAltern (line 6) We now have to assess all of alternatives (parameter *alt_set*), in order to find the solution that fits best to our problem. Therefore, we also need to take into account the tag we assigned to the eR. We know, that the new functionality is not mission-critical. It rather addresses a usability concern.

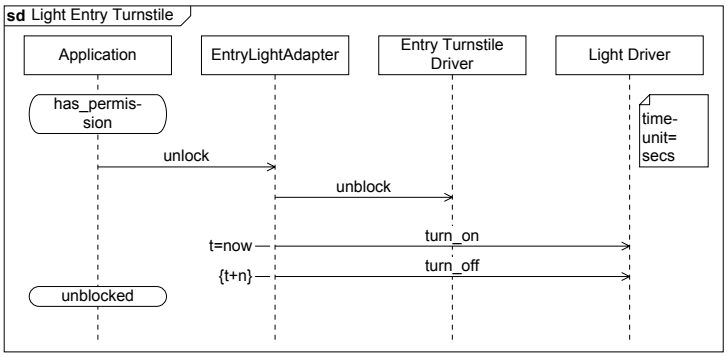


Fig. 8. Internal spec. for Alt. I of Scenario I.

Keeping this in mind, we evaluate the different alternatives:

Alternative I (*Light* grouped with *Entry Turnstile*). This alternative is very intuitive as entry and light serve the same purpose, namely admitting authorized persons to the building. Furthermore, it is not necessary to make any changes to the application as the adapter can handle the new behavior, as well.

Alternative II (*Light* grouped with *Data Base*). Similar to Alternative I, it is possible to group the light with the database as the light relies on a message sent by the database.

In contrast to Alternative I, however, it is not so obvious or intuitive to group the light with the database.

Alternative III (handling *Light* separately). Of course, this alternative provides the most modular and straight-forward solution. In contrast to the two other solutions, it is necessary to modify the application component.

With that, we have assessed all our alternatives (parameter *as_set*).

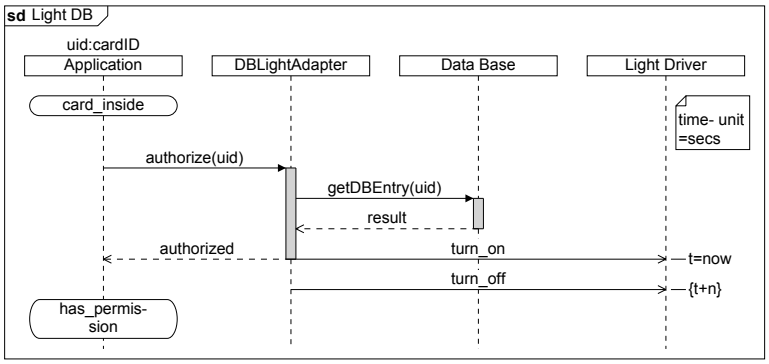
***select**$_{cand}$* (line 7) Based on the previous step, we select the alternative which is suited best for our current situation (parameter *cand*). Here, the experience of the engineers comes into play as the selection will be based on it.

deriveIntSpec (line 8) Usually, only the internal specification for the chosen alternative is derived (parameters *cand*, *spec*). In this paper, we derived specifications for each of the aforementioned alternatives. Figures 8, 9(a), and 9(b), respectively, show the internal specifications for the three alternatives.

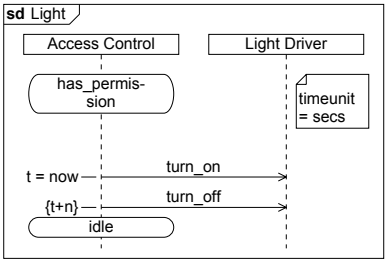
evolve (line 10) Incorporating the new functionality lies beyond the scope of this paper.

5 Related Work

The evolution of software has long been recognized as one of the most problematic and challenging areas in the field of software engineering. Yet, it has received relatively little attention in the past.



(a) Alternative II of Scenario I



(b) Alternative III of Scenario I

Fig. 9. Internal specifications.

Cheesman and Daniels [4] propose a process to specify component-based software, which starts with an informal requirements description and produces an architecture showing the components to be developed or reused, their interfaces and their dependencies. The approach may seem similar to ours. However, our aim is to evolve an existing system and not to specify a new system from scratch.

In an earlier paper [5], we have addressed the problem of adding features to component-based systems. The method described there relies on the process proposed by Cheesman and Daniels [4]. It does not consider architectural patterns. Methodological aspects of evolving a given component-based system to make it more dependable have been introduced in [6]. The approach is to increase the dependability of component-based software by taking exceptional behavior into account. The aim is to preserve the normal cases implemented by the application by intercepting and possibly modifying its inputs and outputs to "shield" it against the exceptional behavior.

Sadou et al. [7] propose a model for software architecture evolution composed of three abstraction levels: the meta level, the architectural level and the application level. It offers a set of evolution concepts, namely operations, rules, strategies and invariants, to describe and manage uniformly the evolution of architectures at the architectural level as well as at the application level. With this approach we share the usage of evolution operations such as addition, deletion, etc. In [8], the authors enhanced their previous

work by enriching the semantic of connectors to be able to determine and propagate the impact of evolutions. The focus of their work, however, lies more with the structure of the architecture. Our approach, in contrast, is aimed at providing different realization alternatives for a given evolution task at the architectural level and then to modify the software according to the chosen alternative.

A framework for designing software architecture through step-by-step refinements is proposed in [9]. The main idea of this project is inspired by the aspect-oriented software development concepts. It provides three main features: a mechanism to add new concerns to a software architecture specification, a description model for software architecture and specific rules, which guarantee the correct integration of a technical concern into a business model. The decision on where to put the new concern is left to the architect in this approach with no further help. Our approach is aimed at deriving different schemata as alternatives for one concern. The architect can then use these schemata to base the decision where to put the new concern.

In the approach proposed by Casanova et al. [10], the authors review the concepts of version and configuration management in order to apply them in the context of component-based software development. The use of multi-dimensional component libraries is proposed as well as a configuration model for component-based applications based on components and connectors. In this way, the libraries not only support component storage and retrieval, but also version and configuration management of components. The focus of this approach is to classify, document, retrieve and version components, not to evolve the software system at the architectural level.

6 Conclusion and Future Work

In this paper, we have described a method that applies schemata and heuristics to guide software engineers through evolution tasks. With this method, it is possible to perform component-based software evolution systematically on the architectural level whenever new requirements or changes in the application environment occur.

As a proof of concept, we evolved an access control system. The evolution was achieved by applying our method step-by-step using the schemata to determine several alternatives for the same evolution task. These alternatives have then been assessed according to heuristics provided by the method to find the solution which fits best under the given circumstances.

In summary, the advantages of our approach are the following:

- It gives guidance on how the addition, modification, or replacement of components, environment or requirements can be performed in a systematic way.
- Using the combination of operator and element, it is possible to classify the evolution.
- Architectural schemata describe how and where the changes have to be implemented.
- It is possible to determine several solutions for the same evolution requirement. Heuristics help to choose from the different alternatives.

We are currently working on a workflow-tool that supports the software engineer in carrying out evolution tasks based on our method.

References

1. Zave, P., Jackson, M.: Four Dark Corners for Requirements Engineering. ACM Transactions on Software Engineering and Methodology 6(1), 1–30 (1997)
2. UML2: Unified Modeling Language: Infrastructure and Superstructure. Object Management Group (2007) Version 2.1.1, formal/07-02-03, www.uml.org/uml.
3. Afadl2000: Etude de cas : système de contrôle d'accès. In: Journées AFADL, Approches formelles dans l'assistance au développement de logiciels, actes LSR/IMAG (2000)
4. Cheesman, J., Daniels, J.: UML Components – A Simple Process for Specifying Component-Based Software. Addison-Wesley (2001)
5. Heisel, M., Souquières, J.: Adding Features to Component-based Systems. In: Ryan, M., Meyer, J.J.C., Ehrich, .D. (eds.) Objects, Agents and Features. LNCS, vol. 2975, pp. 137–153. Springer, Heidelberg (2004)
6. Lanoix, A., Hatebur, D., Heisel, M., Souquières, J.: Enhancing Dependability of Component-Based Systems. In: Abdennadher, N., Kordon, F. (eds.) Reliable Software Technologies – Ada Europe 2007. LNCS, vol. 4498, pp. 41–54. Springer, Heidelberg (2007)
7. Sadou, N., Tamzalit, D., Oussalah, M.: A Unified Approach for Software Architecture Evolution at Different Abstraction Levels. In: IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution, pp. 65–70. IEEE Computer Society (2005)
8. Tamzalit, D., Sadou, N., Oussalah, M.: Connectors Conveying Software Architecture Evolution. In: COMPSAC (1), pp. 391–396. IEEE Computer Society (2007)
9. Barais, O., Lawall, J., Le Meur, A.F., Duchien, L.: Safe Integration of New Concerns in a Software Architecture. In: Proceedings of the 13th International Conference on Engineering of Computer Based Systems (ECBS'06), pp. 52–64. IEEE (2006)
10. Casanova, M., Van Der Straeten, R., Jonckers, V.: Supporting Evolution in Component-Based Development Using Component Libraries. In: CSMR '03: Proceedings of the 7th European Conference on Software Maintenance and Reengineering, p. 123. IEEE Computer Society (2003)