Optimising software development policies for evolutionary system requirements

Joost Noppen, Bedir Tekinerdogan, Mehmet Aksit, Maurice Glandrup & Victor Nicola

TRESE Software Engineering group, Dept. of Computer Science,
University of Twente, P.O. Box 217, 7500 AE, Enschede, The Netherlands

{noppen|bedir|aksit|glandrup|vfn}@cs.utwente.nl

Abstract: Anticipating future software requirements might support the evolution of software systems and as such reduce the cost of development and maintenance in due time. Unfortunately identifying the right set of evolution scenarios is difficult due to the uncertainty of occurrence of future requirements. In this paper we propose the Software Evolution Analysis Model (SEAM) that provides a probabilistic model for evolution requirements, which can more precisely anticipate future requirements and as such reduce the chance on unanticipated requirements. In SEAM the cost of each individual evolution requirement is defined and an optimal set of evolution scenarios is calculated using dynamic programming techniques.

1. Introduction

Software systems are rarely developed for a fixed context and very often the software needs to be changed to fit the new requirements. If these requirements are not anticipated then changing the software system could become cumbersome, expensive, and ultimately untenable. To cope with the changing requirements explicitly anticipating the requirements for possible evolutionary versions of the software system might be a sound option. However, anticipation of requirements might not always be possible or even required. This is because customer requirements are defined by various factors such as market concerns, company growth, future plans, etc. Since these factors cannot be easily controlled within a given time, anticipating the requirements in a deterministic way is likewise practically impossible. Moreover, even when all the possible evolution scenarios can be identified, implementing all the corresponding requirements to anticipate evolution might also not be a viable option, because the software system may become too large and/or the related costs may become too high.

Due to the difficultiy in anticipating evolution scenarios, software engineers might easily miss potential future requirements and/or include requirements that will never be implemented. The first case is an example of unanticipated requirements and might result in a system that cannot even handle these requested requirements. The latter case typically leads to an unnecessary loss of resources.

To reduce the occurrence of these cases a more systematic and precise *evolution analysis* of requirements is needed. For this, we think at least that the following issues should be addressed. First of all, since it is not easy to anticipate the requirements with certainty, the evolution analysis should be able to cope with this uncertainty. Some requirements, may have a very low chance of occurrence in the future, whereas others might be more certain. Secondly, the evolution analysis should be able to select the set of evolution scenarios that perform optimally with respect to the cost of development.

In this paper we present the software evolution analysis model (SEAM), which aims to provide support for deciding which set of requirements need to be involved in future versions of the software system. SEAM provides a probabilistic model of the evolution requirements and in addition integrates the

cost of the various techniques. By applying Markov decision models the optimal evolution scenarios can be more easily determined and as such the initial system can be better prepared for future versions.

The remainder of this paper is as follows. In section 2 we will describe an example case that we will utilise throughout the paper to illustrate our ideas. This example is based on the case described in [1]. Section 3 will present the *software evolution analysis model (SEAM)*. Section 4 will describe the related work. Finally, section 5 will present the conclusions.

2. Example: Email system

Consider a simple mail system, which consists of classes Originator, Email, MailDelivery and Receiver. As an example, the interface methods of class EMail are shown in the following figure (declared in a generic OO-style language):

```
Class Email interface
putOriginator(anOriginator);
getOriginator returns anOriginator;
putReceiver(aReceiver);
getReceiver returns aReceiver;
putContent(aContent);
getContent returns aContent;
send;
reply;
approve;
isApproved returns Boolean;
putRoute(aRoute);
getRoute returns aRoute;
deliver;
isDelivered returns Boolean;
```

Figure 1. Interface specification of class Email class

EMail represents the electronic messages sent in this system and provides methods for defining, delivering and reading mails. For example, the methods *putOriginator, getOriginator, putReceiver, getReceiver, putContents, getContents* are used to write and read the attributes of a mail object. The methods *putRoute, getRoute, deliver, isDelivered* are used by class MailDelivery while delivering the messages from originators to receivers. The method *reply* is used to send a reply message. In this paper, EMail will be used as the base class that will be evolved with new requirements.

3. Software Evolution Analysis Model (SEAM)

In this section the SEAM approach will be described, which tries to solve the problem of finding the best policies for developing the software system. For this SEAM should consider the following variables:

- Possible future changes
- Probabilities of these changes
- Techniques for dealing with these changes
- The cost of applying these techniques from a certain situation

To do this SEAM consists basically of three main processes: *feature modelling, modelling evolution scenarios,* and *analysing the evolution scenarios using Markov decision models*. We will explain these processes in the following sub-sections.

3.1 Feature model

The problem of software evolution starts at the requirements modelling level. Because the new requirements of the software system affect the existing requirements, it is important to know how these requirements relate to each other.

To define the set of requirements we apply feature modelling as it is applied in domain analysis. Hereby, features are defined as *"the attributes of the system that directly affect end-users"* [2]. A *feature diagram* defines the domain model with the corresponding features of a system and their relations. Usually, the modelled features define the properties of one or more systems sharing the same semantics. Hereby, less attention is given to future requirements and only those features are included that are basically needed for the initial development of one or more systems. We extend the use of feature models and also

include features that are needed in future versions of the system(s).

Now assume that there are two evolutionary requirements for the Email system.

First, like in a postal mail system, we would like to restrict accesses to email objects based on the type of the client object. If the client is of the *user* type, it is allowed to execute the methods *putOriginator*, *putReceiver*, *putContents*, *getContents*, *send* and *reply*. The methods *approve*, *putRoute* and *deliver* are used by the clients of the *system* type. No restrictions will be defined for the methods *getOriginator*, *getReceiver*, *isApproved*, *getRoute* and *isDelivered*.

Second, to cope with various data we would like to be able to adapt the implementation of the method *send*. For instance the protocol for sending an email may be changed in case a video is being sent. This means the system has to detect the type of content that should be sent, and select the protocol that is suited best to send the attachment.

To represent these evolution scenarios Figure 2 presents the feature model for the EMail system. In this feature model we can identify the two evolutionary features for the e-mail system: *USViewMail* and *DynamicMail*. Both features can be seen as independent extensions to the Email system. For both *USViewMail* as well as *DynamicMail* the Email system needs to be implemented, but *USViewMail* doesn't need *DynamicMail* and vice versa.

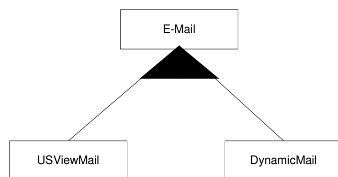


Figure 2. Feature model for E-mail system

3.2 Requirements evolution diagram

At least as important as the possible changes that will be included is the order in which these changes can be demanded. This might have a direct impact on the maintenance and the cost of the software system. To cope with this issue the scenarios that are possible with the known possible changes will be modelled.

An evolution scenario consists of several steps, where each step represents a single requirement change. When a change occurs the system is changed accordingly which leads to a new software system on which new evolution requests can be applied. Such an evolution scenario can be modelled as a state-transition diagram. A state in the diagram represents the current system and a transition represents the event that the customer changes the requirements of the current system.

Several evolution scenarios can share one or more states with other evolution scenarios for a software system. And since the start-state will be the same for all of them they can be modelled in one state-transition diagram, which includes all possible evolution scenarios for known changes. This state-transition diagram will be called a *requirements evolution diagram*.

The states of the requirements evolution diagram are derived from the nodes (features) of the feature tree. This has the consequence that the relations in the feature tree also restrict the order in which the state transitions in the requirements evolution diagram can occur.

Not every evolution scenario will occur with the same probability. Depending on the environment, market changes, etc. a customer might prefer one feature to another. This uncertainty can be introduced into the evolution scenarios by applying a probability model to the transitions in the requirements evolution diagram.

Each transition represents the event that the customer asks for a feature from the set of features that has not yet been asked for. By assigning a probability to every transition from certain state, the uncertainty of evolution scenarios can be modelled.

An *evolution scenario* is defined as a path from a requirements evolution diagram. A path consists of a set of states and the corresponding transitions herein. It is possible to derive multiple evolution scenarios from the same evolution diagram.

Email example

Figure 3 represents the requirements evolution diagram for the Email system. In this diagram three states can be reached from the initial state: *U-S*, *None* or *Dyna.* The states *U-S* and *Dyna* represent the features *USViewMail* and *DynamicMail* respectively, as defined in the

feature diagram of Figure 2. The state *None* represents the event(s) in which no evolution is required anymore.

When we assume that the order in which the requirements can occur is irrelevant for this example, from the state *U-S* as well as the state *Dyna* it is possible to continue the evolution to the state *Both,* which corresponds to both evolution scenarios. The state *No More* defines the case in which after one evolution step no more evolutions are required anymore.

Since the evolution scenarios cannot be anticipated with certainty each transition in the evolution diagram has been assigned a probability value. Hereby, the output of the sum of probabilities from one state equals one. Because of the probability values an important part of the unanticipation has been characterized.

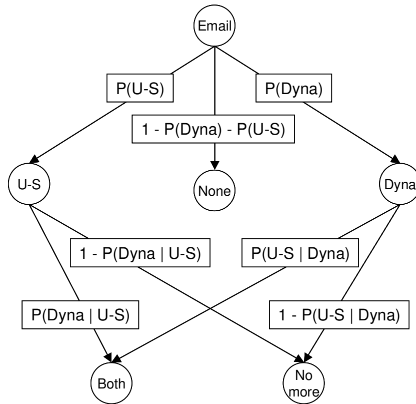


Figure 3. Evolution diagram for the e-mail system

3.3 Markov decision process formalism

Recall that every evolution diagram includes multiple evolution scenarios. Each of these evolution scenarios may have a different impact on the software system and as such require a different way of addressing the evolution. It is now possible to select the evolution scenarios with the highest probability of occurrence. However, besides of the probability of occurrence also the cost of development usually plays an important (sometimes decisive) role.

An evolution scenario with a low probability might lead to very high costs if it is not prepared for, although later on the features in the scenario are still needed. In that case preparing the system also for features with a low probability might still be a better option. On the other hand an evolution scenario with a high probability may have a very high cost and as such might be reconsidered for selection.

Costs should therefore also be included in the selection process on which evolution scenarios should be supported. The choice of one preparation technique over another may lead to different costs, and the cost reduction at the moment evolution occurs can also vary. A mechanism is needed that can optimise the choices for preparation techniques with respect to costs and probabilities of evolution scenarios.

In SEAM a Markov decision process formalism is defined which makes it possible to identify the best techniques based on costs and probabilities of software evolution scenarios.

From the engineer's point of view the decision on how to prepare the system for the possible requirements changes should be done from the current state in the evolution scenario. The decision leads to implementation using a certain technique which will have a certain cost. Depending on which evolution scenario will unfold, the preparation technique will be able to reduce costs or not.

This type of problem resembles the theory of sequential decision problems closely. In SEAM a class of sequential decision problems called Markov Decision Problems, is used to model the decision process that occurs for a software engineer when a software system should be prepared for the consequences of software evolution. The formalism of Markov Decision Problems is based on finite state automata, where in each single state a decision has to be made on which transaction should be chosen. Such as state automaton is called a Markov Decision Problem. The theory of Markov Decision Problems also describes solving algorithms and optimisations. Numerous good textbooks on Markov Decision Problems and Dynamic Programming exist, like for instance [2].

The decisions on which technique to include are done at every stage in an evolution scenario. After the decision has been made the evolution scenario will continue, which means changes to requirements might occur or not.

When this is modelled with Markov Decision Problems this means that a stage at which a decision has to be taken is represented by a state. From such a state several actions can be chosen, which in this case will be the decision on any of the available techniques. The states in the Markov Decision Problem represent the possible stages of the evolution scenario. This makes it possible to use the requirements evolution diagram as the base for the Markov Decision Problem.

When several preparation techniques have been identified, from every state (except end-states) in the requirements evolution diagram any of these techniques can be chosen. For every single technique the same features can be demanded by the customer. Note that states in a Markov Decision Problem with equal feature sets are different for every technique. For the Markov Decision Problem the technique that is chosen in the previous state can influence future efforts. This means two states with equal feature sets, but with different techniques are not considered equal.

In SEAM a state in the Markov Decision Problem that represents the evolution process is a tuple. This tuple is defined in the following way:

State$_x$: (Π, δ, Ω)

Π	=	$\{ f_1, f_2, \ldots, f_n \}$
δ	=	f_x
Ω	=	$\{ T_1, T_2, \ldots, T_m \}$

Π represents the set of features that already have been implemented in the current state of the evolution process. This set can be ordered or unordered, depending on whether or not this is relevant to the analysis. δ represent the feature that has been asked for by the customer and still has to be implemented. Ω represents the set of techniques that have been used up to now to implement the system.

Depending on the state configuration the current situation in the evolution process can now be expressed in terms of features and techniques. This resembles the way in which the engineer experiences the optimisation process closely. At a certain point in the evolution process a system engineer has to decide whether or not to prepare a system for pssobile future requirements. This is usually when a new requirement occurs. At this point a decision has to be made on how to implement the new requirement and evolution preparation. The process then enters a wait-state until the next requirement occurs.

An action in SEAM represents the event of choosing a technique for supporting software evolution. Each such an action can have different outcomes, representing the uncertainty of software evolution.

When the requirements evolution diagram is taken as the base for the Markov Decision Problem a Cartesian product needs to be performed to represent all the possible configurations that can occur (each state in the requirements evolution diagram can occur for every single preparation technique or combination). The following methodological approach can be applied for defining the Markov Decision Problem from a requirements evolution diagram:

1. Set the start-state of the requirements evolution diagram as the start-state of the Markov Decision Problem;
2. The actions that can be chosen from this state are the techniques that have been identified;
3. For every state that can be reached from this state in the evolution diagram, a corresponding state is made for every technique (except for end-states);
4. The next-states are added to the possible next-state set of the actions of this state;
5. Step 2, 3, 4 and 5 are repeated for the states that have been generated in the Markov Decision Problem until no more new states occur;
6. Add the total end-state to the Markov Decision Problem.

In the theory of Markov Decision Problems for each state an optimal action (decision) is determined based on *reward functions*. Such a reward function returns a reward depending on the action that is chosen at a specific state. Reward functions are normally defined in the following way:

$$Reward(S_t, a_t)$$

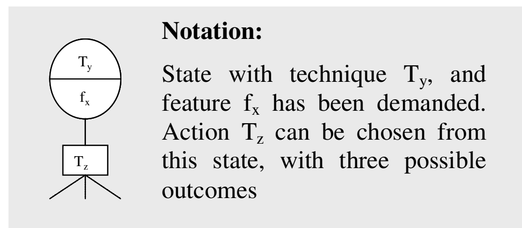
The function *Reward* will return the cost of choosing action a_t in state S_t. Optimisation solutions for Markov Decision Problems are aimed at minimising (or maximising) the total value of each of these contributions by selecting the best decision for each state.

In SEAM *reward functions* are used to define the cost of implementing the known requirements at that stage with the technique that has been chosen. Based on the information from the state (which features have been asked for, which technique(s) ha(s)(ve) been used earlier, etc.) and the technique that is chosen, the reward function can calculate the contribution. By applying the optimisation solution algorithms to the Markov Decision Problem that represents the software evolution, and minimising costs, this results in the decisions for each state for which minimal costs will occur (when the Markov Decision Problem is deterministic) or can be expected (when the Markov Decision Problem is stochastic).

The way in which these costs are calculated can be based on any existing or newly defined cost model that returns accurate figures for choosing techniques. In this paper a very naive and simple cost model is adopted to demonstrate SEAM, but this is a main part of future research.

Email example

For the Email example a simple cost model is defined in *figure 5*, which makes the order in which the features occur irrelevant (because the costs will be equal). For this example the following notation will be used for representing states and actions:



In this notation a state is depicted by circle, that is divided in half. The top half displays the technique that has been used to implement the system up to now (only one technique is possible because for this example the techniques are assumed to be mutually exclusive). The bottom half contains the feature that has now been demanded by the customer but still needs to be implemented.

An action in SEAM is depicted by a box. This box represents a choice for a certain technique. Once this choice has been made, several different states can be reached, each with its own probability. Ths represents the probabilistic character of the software evolution process.

For the Email system already a requirements evolution diagram has been defined. When it is assumed that two different, mutual exclusive techniques need to be assessed, *Inheritance* and *Composition Filters*, this would lead to the Markov Decision Problem in Figure 4.

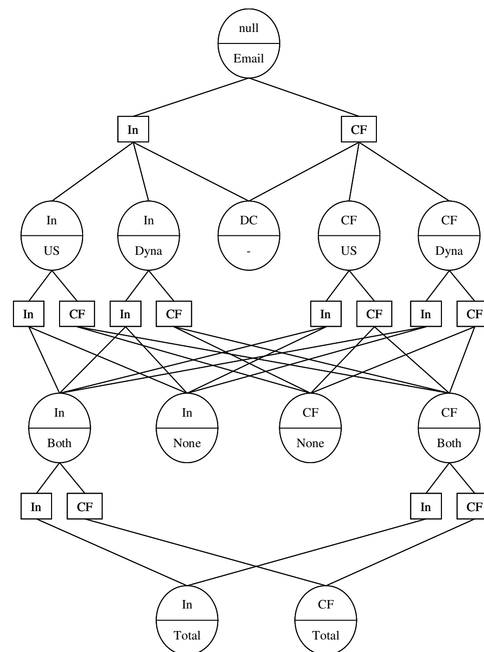


Figure 4. Markov Decision Problem for the e-mail system evolution

First the start-state of the evolution diagram (Email) is identified as the start-state *(null | Email)* of the Markov Decision Problem. Because two techniques have been identified these will be the possible actions from the start-state. The third step defines the states that can be reached from this state. In the evolution diagram three different states can be reached: *U-S, Dyna* and *None*. First *U-S* needs to be defined for the different techniques: the states *(In | US)* and *(CF | US)* are defined and assigned to the possible next states set of *Inheritance* and *Composition Filters*

respectively. For the state *Dyna* the following states can be calculated: *(In | Dyna)* and *(CF | Dyna)*.

The third state that can be reached, *(DC | -)*, is an end-state (DC means *Don't care*, and no feature has been asked for). This means the software evolution process will stop (at least for this analysis). It can be included directly in the possible next-states set of both techniques.

This process now is repeated for the states that have been defined in the first iteration. The techniques once again are *Inheritance* and *Composition Filters*, which means the actions are the same as the previous state. The corresponding state (the state without the technique) in the evolution diagram for state *(In | US)* is *U-S*. The possible next states for this state therefore should be based on *Both* and *No more*. Once again states need to be added for every technique: *(In | Both)* and *(CF | Both)*. These states are added to their respective techniques. The end-state is done in the same way as the previous iteration but here the distinction is made between the techniques that have been used. Whether or not this should be included depends on the relevance for the analysis. Repeating this process leads to states that are equal to the states that have been defined by *(In | US)* in this case. This completes the third level.

After this no new states occur in the Markov Decision Problem. At this point the total end-states need to be added. The total end-states can be reached from any state where all possible features have been demanded, in this case *(In | Both)* and *(CF | Both)*. As was mentioned before a state in the Markov Decision Problem represents a system where one feature still has to be implemented. This can be done using either technique *Inheritance* or *Composition Filters*, but the result will always be the same from a software evolution perspective: the evolution is completed. This is represented by the total end-states (In | Total) and (CF | Total).

Now the entire software evolution process is modelled, the techniques can be assessed for their effectiveness. Figure 5 defines a very simple cost- and probability-model for the Email system:

	Costs		
Technique	**Email**	**User-System**	**Dynamic**
Inheritance	65	65	65
Comp. Filt.	55	55	55

Probability	**Value**
P(U-S)	0.45
P(Dyna)	0.45
1 - P(U-S) - P(Dyna)	0.10
P(Dyna | U-S)	0.45
P(U-S | Dyna)	0.45
1 - P(Dyna | U-S)	0.55
1 - P(U-S | Dyna)	0.55

Figure 5. Simple cost model for e-mail system .

This cost model assumes that the implementation of each feature will be equal for a single technique. This means that the implementation of for instance DynamicMail using Inheritance will cost 65. Note that this is only the case when this is done from the situation that the existing system is implemented using inheritance structures. When this is not the case the entire system needs to be reimplemented because of the *mutual exclusion* of Inheritance and Composition Filters.

Now for each state the expected minimal cost for each action can be calculated, based on solution algorithms that have been defined for Markov Decision Problems. At the *Total*-states no actions can be chosen and therefore the reward for this state is set to 0.

The state *(CF | Both)* has two possible actions: Inheritance and Composition Filters. When Composition Filters is chosen, this action would cost 55 because a new feature has to be implemented in a system that already uses Composition Filters. Inheritance would cost 195 because two features have to be reimplemented in addition to the new feature (65 + 65 + 65). The second part of the costs that needs to be added is the cost that can be expected from the states that can be reached. For both techniques this is 0 (from the *Total*-state). This means the minimal expected costs

for Inheritance is 195 and for Composition Filters 55. For the state *Both,Inheritance* the computation is equivalent and results in 65 for Inheritance and 165 for Composition Filters. Note that all the information that is needed for this computation is included at the current state: the techniques that were used, the features that already have been implemented and the feature that still needs to be implemented. Information concerning mutual exclusion of techniques is defined in the cost function, to ensure a proper separation of concerns.

When Composition Filters are chosen for the state *(CF | Dyna)*, 55 is awarded by the reward function. But from this action two different states can be reached: *(CF | Both)* with probability 0.45, and *None* with probability 0.55. The expectation value of these two states should be added to the total expected costs of choosing Composition Filters:

55 + (0.45 * 55) + (0.55 * 0) = 79.75

For Inheritance this would be:

65 + 65 + (0.45 * 65) + (0.55 * 0) = 159.25

This would leave Composition Filters as the best action. This can be done for every single state in the Markov Decision Problem, which would lead to the following results:

State	**Best Action**	**Exp. Cost**
(Null | Email)	**Comp. Filters**	**126.775**
(In | US)	Inheritance	94.25
(In | Dyna)	Inheritance	94.25
(CF | US)	Comp. Filters	79.75
(CF | Dyna)	Comp. Filters	79.75
(In | Both)	Inheritance	65
(CF | Both)	Comp. Filters	55

Figure 6. results of the Markov Decision Problem

From the table it can be seen that Composition Filters would be the best choice for the initial state of the Email system (Inheritance has an expected cost of 149.825).

4. Related work

In this section we briefly point to a selection of related work.

The analysis of software systems that are related to the system that should be made is done by several problem domain engineering methods, of which FODA [2] is one. Most of these methods are aimed at understanding the intrinsic difficulties of typical systems in the problem domain. Most of the typical characteristics of software evolution (e.g. uncertainty) cannot be modelled by domain engineering methods.

Product lines [4] is a domain engineering technique that focuses on common properties of software systems in a problem domain. Based on these commonalities a base is defined on which equivalent systems can be based. The method does not offer support for uncertain requirements changes however. These changes could lead to an invalid product line base, because commonalities that have been identified earlier could be cancelled out. It seems a proper scoping of the product line becomes very important, because software evolution might lead to the fact that the asset base shifts out of scope, and new assets and thus new product families need to be defined.

G. Kahen, M.M. Lehman, J.F. Ramil and P. Wernick have done extensive research on the Software Evolution phenomenon. The have identified the software evolution process to be a feedback system where each step in the evolution influences the way in which the process will behave in subsequent steps. By dividing software systems into three different categories the different types of evolution can be categorised. Each system will be subject to evolution scenarios of different character, depending on the type of system, and the environment it operates in.

This research lead to the postulation of the five laws of software evolution, to which later another three were added. This research was continued in the FEAST/1 and FEAST/2 projects. As opposed to SEAM this research is aimed at defining the dynamics of the software evolution process inside the software lifecycle, whereas SEAM is aimed at finding optimal decisions at specific (and relevant) points of likely evolution scenarios that could occur during the lifetime of a software system.

Cook, Ji and Harrison [5] proposed to address the phenomenon of software evolution by measuring evolvability at different levels of abstraction. The metrics that are used and their

results can enhance the insight in the way software evolution of a particular system and its environment interact. A first attempt was made to model the occurrence of new requirements by using a probabilistic queueing model. The accuracy of this model seemed to improve after reconfiguration for specific projects that were analysed. This seems to coincide with the statement of Lehman that software evolution should be seen as a complex feedback system.

This approach resembles SEAM partially, because SEAM addresses software evolution on one of the abstraction levels that is mentioned. The feedback that was missed when the queueing model as applied can be included in SEAM by using Markov Decision Problems that incorporate "learning" states. The optimal decision than depend not only on the current state, but also what was learned from the route leading up to this state. The same queueing model could perhaps be used as the stochastic model for evolution uncertainty.

5. Discussion and conclusion

In this paper a software evolution analysis model is presented with which it is possible to find the best development policies for evolutionary requirements. This model can be used for analysis of software evolution to reduce costs for upgrading existing software systems.

By using the FODA feature model the relations between the customer requirements can be modelled, and based on these relations the possible evolution scenarios can be defined. These evolution scenarios can be analysed for their impact and by adding probability models to the scenarios, the uncertain character of software evolution can be modelled.

The relevant events of software evolution, the customer asking for changes and the engineer deciding on how to solve the problems, are related to each other with Markov Decision Problems. This theory can accurately capture the decision problem an engineer faces when working with software evolution. By defining an accurate cost model the results of decision made during a software evolution process can be optimised.

The effectiveness of this approach is dependent on several issues. First of all, to get valuable feedback the set of identified changes should contain the most relevant evolutions. When this is not the case, software evolution might still cause costs of maintenance to explode. To improve on this part further research is needed on market analysis and other sources that might lead to the identification of possible system changes.

When evolution is truly unanticipated it is not prepared for by definition. And possible generic preparation methods that address some issues might affect others (e.g. the occurrence of non-functional requirements changes when functional requirements changes can be dealt with). It seams software evolution causes a shift of scope, an not anticipating for this means the scope of the relevant domain is exceeded in an unexpected direction. The amount of unanticipation is then determined by how close a border is reached of a problem domain that was not analysed in any way.

Sometimes it is possible to know what is not known. This could be a black-box approach where the borders of the unanticipated evolution are known, but within these borders the evolution is unanticipated. For instance when the protocols for communication are unknown, it might be possible to define a protocol-generating protocol. These kinds of techniques can be included as design alternatives in SEAM, thus making the model act on a higher level of abstraction.

An important issue is the complexity that is caused by this type of analysis. By using Markov Decision Problems for representation of the software evolution process this can lead to a *state-space explosion* quite easily. To represent all possible situations a large amount of states is needed .

Furthermore the probability models can influence the relevance of the feedback given by SEAM. The current model uses discrete stochastic modelling but it is more likely the probabilities for customer asking for system changes are dependent also on time. This means a continues stochastic model will be needed, which is subject to further research.

Finally the cost modelling is of major influence on the effectiveness of SEAM. When reward functions accurately model the

costs of choosing techniques, SEAM can provide accurate feedback on this subject. For this standard cost models can be used, but for a more accurate way of modelling costs further research is needed.

References

[1] Aksit, M., Bergmans, L., *Software evolution problems in case of inheritance and aggregation based reuse*, 2001, e-tutorial downloadable from: http://trese.cs.utwente.nl/

[2] Kang, K. e.a *Feature-oriented domain analysis (FODA) feasibility study*, SEI Interactive, Carnegie Mellon University. PS-file: FODA.ps from http://interactive.sei.cmu.edu/

[3] Kahen G., Lehman M.M., Ramil J.F., Wernick P. *System dynamics modelling of software evolution processes for policy investigation: Approach and example*, Department of Computing, Imperial College of Science, Technology and Medicine UK. Elsevier Science 2001.

[4] Software Engineering Institute, *A framework for software product line practice - version 3.0*, Carnegie Mellon University, http://www.sei.cmu.edu/plp/frame_report/coreADA.htm

[5] Cook S., Ji H., Harrison R. *Software evolution and evolvability* University of Reading, UK

[6] Puterman, Martin, *Markov decision processes, discrete stochastic dynamic programming*, 1994, Wiley-Interscience, ISBN: 0-471-61977-9