From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process

Manuel Brandozzi
UT – ARISE
Advanced Research in Software Engineering
The University of Texas at Austin
manuelbr@mail.utexas.edu

Dewayne E. Perry
UT – ARISE
Advanced Research in Software Engineering
The University of Texas at Austin
perry@ece.utexas.edu

Abstract

The step from the requirements for a software system to an Architecture for the system has traditionally been the most complex one in the software development process. This step goes from what the system has to achieve, to how it achieves it. In order to make this step easier, we propose the use of Preskriptor, a prescriptive architectural specification language, and of its associated process, the Preskriptor process. Architectural prescriptions consist in the specification of the system's basic topology and of the constraints associated with it and its components and interactions. The Preskriptor process provides a systematic way to satisfy both the functional and non functional requirements from the problem domain, as well to integrate architectural structures from solution domains.

1. Introduction

The most difficult transition in the development process for a non-trivial software system is likely the one from the requirements for the system to the system's architecture. This step involves going from the problem's domain to the domain of its solution [1]. One of the factors that makes the design of software systems so challenging is that they have to satisfy many different requirements (problems) at the same time, and there is often more than a single solution to a particular requirement.

Requirements specifications can be viewed as a contract between the customer and the software developers. Hence, they should be not only easy to understand by the software architects and engineers but also by the domain experts and users.

We propose the use of architectural prescriptions [2] to perform the step from requirements to architecture. An architectural prescription is the architecture of the system in terms of its components, the constraints on them and the interrelationships among the component (i.e., the constraints on their interactions). At least initially, the constraints are only those coming from the problem domain. While architectural descriptions provide more or less complete details to the designers, prescriptions make the step from requirements to architecture easier to model and to perform. Prescriptions may also provide a means of deeper understanding about the architecture. We will show how we can perform this step from goal-oriented requirements. Another advantage of prescriptions is that, being at a higher level of abstraction, they can be reused more easily, and they enable more creative designs.

The same prescription could be used for an entire software family [3] of applications that differ only in deployment requirements. If the applications differ also in some requirements coming from the problem domain, like the interaction with different types of users, we can first develop the prescription for an ancestor system that has all and only the requirements common to the whole family and then get, by extending this prescription, the prescriptions for all the descendent applications.

Because Architectural Prescription Languages APLs, which we introduced in [4], are written in an elementary ontology, they enable new, innovative designs. Let's consider, for example, a distributed system. An architecture description language may include elements such as clients and servers. It may be that the architect writing a specification in such an architecture description language uses client and server components also when, for example, a multi-peer architecture might be a better solution. The designer will then be constrained by such architecture to a low-level design that adopts a client-server solution. By describing the system at a higher level of abstraction, a specification in an architectural prescription language would instead permit the designer to choose the best solution at the design level and even let him/her take different choices for different members of the family.

The paper is structured as follows: we first give an overview of KAOS, the requirements specification language our process uses as a starting point; then we introduce the Preskriptor architectural prescription language and process illustrating them with a practical example; we conclude by summarizing the fundamental

results of the paper, and by discussing the future directions of our research.

2. Overview of the KAOS Specification Language

KAOS is a goal oriented requirements specification language [5]. Its ontology is composed of objects, operations and goals. Objects can be agents (active objects), entities (passive objects), events (instantaneous objects), or relationships (objects depending on other objects). Operations are performed by an agent, and change the state of one or more objects. They are characterized by pre-, post- and trigger- conditions.

Goals are the objectives that the system has to achieve. In general, a goal can be AND/OR refined till we obtain a set of achievable sub-goals. The goal refinement process generates a goal refinement tree. All the nodes of the tree represent goals. The leaves may also be called requisites. The requisites that are assigned to the software system are called requirements; those assigned to the interacting environment are called assumptions.

Let's briefly see how obtain a requirements specification in KAOS. The high-level goals are gathered from the users, domain experts and existing documentation. These goals are then AND/OR refined till we derive goals that are achievable by some agents. For each goal the objects and operations associated with it have to be identified. Of course, more than one refinement for a goal may be possible, and there may be conflicts between refinements of different goals that can be resolved as proposed in [6]. It's up to the requirements engineer to generate a "good" refinement tree. By "good" refinement tree we mean one that does not contain conflicts among refinements of different goals and from which it is possible to derive an architecture that achieves those goals. In addition to iterations with the requirements specification process, there may also be iterations between the requirements specification process and the architecture prescription process.

In figure 1., there is an example of a goal specified in KAOS, taken from the example we'll use in next section.

Goal Maintain[ConfidentialityOfSubmissions]
InstanceOf SecurityGoal
Concerns DocumentCopy, Knows, People
ReducedTo
ConfidentialityOfSubmissionDocument
ConfidentialityOfIndirectSubmission
InformalDef A submission must remain confidential. A paper that has to be submitted has to remain confidential.

Figure 1. Example of a goal specification in KAOS

The keyword *Goal* denotes the name of the goal; *InstanceOf* declares the type of the goal; *Concerns* indicates the objects involved in the achievement of the goal; ReducedTo contains the names of the sub-goals into which the goal is resolved. *InformalDef* is the informal definition of the goal. Then there could be *FormalDef*, n optional attribute; it contains a formal definition of the goal (which can be expressed in any formal notation such as first order logic).

3. The Preskriptor Process

We will illustrate our technique with an example. In the example, we shall obtain an architectural prescription for a system that automates some of the functions in the paper selection process for a scientific magazine (or a conference). Our starting point is a specification of this software system in KAOS. The fundamental goal of the paper selection system is to keep high the quality of the magazine.

We have to determine the fundamental goal (root goal) that the system has to achieve; this goal is the only unavoidable constraint coming from the problem domain. By using a KAOS specification as a starting point, we can gradually increase the degree of constraint of the solution by considering the goals that refine the root goal. We can keep on refining goals to an appropriate level. The Preskriptor process can take as input goals in any level of the resulting goal refinement tree.

If we take the root of the tree, although the resulting prescription will enable new, innovative solutions to the problem, it will generally provide too little guidance to the system's designers.

On the other hand, taking the leaves of the goal refinement tree (or even a further refining of the prescription to achieve qualities as performance, reusability, etc.) may produce a specification that constraints too much the lower level designs. As Parnas once noted, if in order to design washing machines we used all the requirements coming from how we wash the clothes by hand, we wouldn't have got the very effective rotary washing machines of nowadays.

Our approach leaves the software architect free to choose the degree of constraint desired on the architecture. Also, he or she could change the degree of constraint during the architecture process according to necessity. In the example that follows we use a high degree of constraint (i.e. we consider goals deep in the goal refinement tree) only for demonstration purposes.

The process of deriving the prescription is composed of three steps that can be followed by an optional one, and which may be iterated. In the first step we derive the basic prescription from the root goal for the system. This root goal is either already given or it can be obtained by abstracting its sub-goals. In the second step we get the

components that are potential sub-components of the basic architecture considering the objects that are in the KAOS specification. In the third step we choose a level of refinement of the goal refinement tree that we consider appropriate, we decide which of the sub-goals at this level are achieved or co-achieved by the software system, and we assign them to the sub-components which we derived at step 2. As a last step, the architectural prescription may be further refined to achieve additional non-functional properties.

Our example considers the KAOS specification for the paper selection process developed in the thesis [7]. We shall transform this KAOS specification into a prescription for a Software System that is to assist in the paper selection process. Figure 2. illustrates the first three steps of the process.

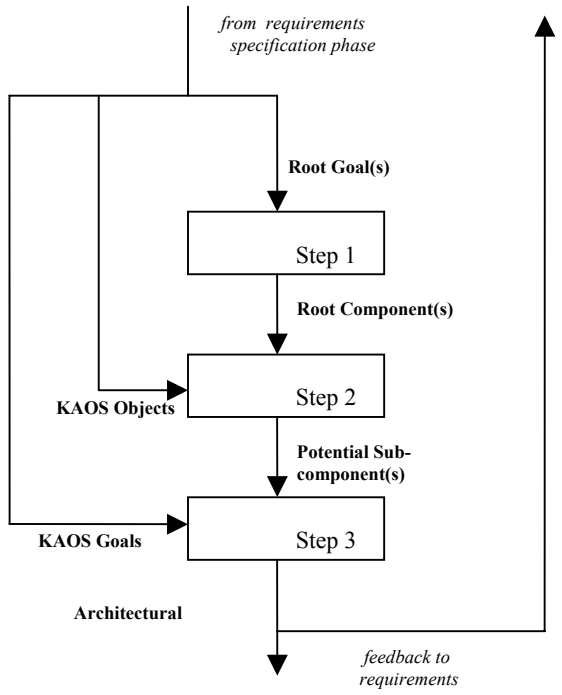

Figure 2: The fundamental steps of the Preskriptor process

3.1 The First Step of the Methodology

The software system, that we hereafter denote as "SelectionManager", is co-responsible for the root goal "Maintain[QualityOfTheScientificMagazine]" together with the system composed of the people involved. The software system performs different functions that can be automated and it interacts with the human system. Its purpose is to speed up the paper selection process and to improve its confidentiality.

The Preskriptor language is an implementation of the APL introduced in [4].

Preskriptor Specification: ScientificPaperManager
KAOS Specification: PaperSelectionProcess
Components:

Component SelectionManager [1,1]
Type Processing
Constraints
Maintain[QualityOfTheScientificMagazine]
Composed of …
Uses PeopleConnect *to interact with* (AutorAgent, ChiefEditorAgent, AssociatedEditorAgent, EvaluatorAgent)

Figure 3: Example of a specification in Preskriptor

At the beginning of a Preskriptor specification is the declaration of its name. It's followed by the declaration of the KAOS specification from which the prescription is derived. A prescription may derive from only one KAOS specification, and if the prescription derives from several different KAOS specifications, it's better to merge the specifications first and then to architect the system. By doing so, if there are conflicts between goals in different specifications they will be solved early at the requirements phase. So, all the components of a prescription derive from the same KAOS specification, which may be the union of several KAOS specifications. Following are the definitions of the components.

The field *Component* specifies the name of the component. *Type* denotes the type of the component. *Constraints* is the most important attribute of a component. It denotes which are the requirements that the component is responsible for. We use here the term constraint to denote both functional and non-functional constraints (both corresponding to requirements on the system). *Composed of* identifies the subcomponents that implement the component. The last attribute, *Uses*, indicates which are the components used by the component. Since interactions can only happen through a connector, the *Uses* attribute has the additional keyword *to interact with* denoting which components the component interacts with using a particular connector.

At the highest layer of abstraction, to which the first step of the specification corresponds, we have to write next to the name of a component its possible number of instances in the system. At the other layers this information is optional because it will be contained anyway in the *Composed of* field of the super-component of the component considered. For example, [1,n] means that the component can have any number of instances from 1 to an arbitrary number n.

We will fill in the *Composed of* field after we decide how to refine the system at the third step. The software

system has to interact with the people involved in the process. To do so, it uses the (fairly complex) connector "PeopleConnect". To distinguish the people involved in the process (agents) from the data components that may be used in the software system to represent them, we added the Agent suffix to their names. PeopleConnect is specified as follows:

Component PeopleConnect [1,n]
Type Connector
Constraints
  Maintain[QualityOfTheScientificMagazine]
Composed of …
Uses /

Figure 4. Example of a connector specification

The symbol "/" means none and, for now, we will omit the fields whose value is none. The formal specification of the Preskriptor language is in the Appendix.

3.2 The Second Step

From the objects in the KAOS specification we derive potential data, processing and connector components that can implement SelectionManager. If in the third step we don't attribute any constraint to these potential components, they won't be part of the prescription. In that case, in fact, they won't be necessary to achieve the goals of the KAOS specification. In figure 5. is a sample this set for the paper selection process.

Component Document
Type Data
Constraints …

Component Paper
Type Data
Constraints …

Component People
Type Data
Constraints …

Component Knows
Type Data
Constraints …
Composed of People[0,m], Document[0,n]

Figure 5. Sample of potential components for the paper selection system

The notation, used in the *Composed of* field of the last component, means that the component is composed of 0 or more "People" sub-components and by 0 or more "Document" sub-components. Obviously, the number of instances assigned to different sub-components doesn't have to be the same.

"SelectionManager" could be composed also of the following processing component, and the following connectors, which connect the processing component to the data ones.

Component SelectionManagerEngine
Type Processing
Constraints
  Maintain[QualityOfTheScientificMagazine]
Composed of …
Uses
  PeopleConnect *to interact with*
    (AuthorAgent, ChiefEditorAgent,
     AssociatedEditorAgent, EvaluatorAgent),
  Conn1 *to interact with* Document,
  Conn2 *to interact with* Paper,
…

Component Conn1
Type Connector
Constraints …

…

Figure 6. SelectionManagerEnging and associated connectors

3.3 The Third Step

Now we will complete the architectural prescription by taking into account the goals that are at the goal refinement tree level that we selected. We show how to put constraints on the architectural components we got at step 2.

Let's first refine our root goal. After a first refinement, the subgoals of the root that the software system needs to achieve are:

Maintain[OriginalityOfSubmission],
Maintain[QualityOfPublishedArticles],
Maintain[QualityOfPrint],
Achieve[EnoughQuantityOfPublishedArticles].

By refining the first of these goals, we obtain the following sub-goals:

Maintain[QualityOfEditorialDecisions],
Maintain[PertinenceOfPublishedArticles].

After two more refinements we obtain:

Avoid[ConflictOfInterestsWithAssociatedEditor]

This goal can translate directly into a constraint on the "SelectionManagerEngine" and "People" subcomponents. "SelectionManagerEngine" will somehow keep track of the different ways the various people represented by the People data component may know each other. The two

constrained components are able to achieve this requirement and the existence of this requirement is a sufficient condition for the existence of the two components given our architectural rationale. By this we mean that these components ought to exist even if they have no other goals to achieve. On the other hand, if we don't care anymore about this requirement and there are no further constraints assigned to these components, there is no point in keeping them. By proceeding in a similar fashion with the rest of the goal refinements, we obtain the first version of a complete Preskriptor specification:

Preskriptor Specification: ScientificPaperSelector
KAOS Specification: PaperSelectionProcess
Components:

Component SelectionManagerEngine [1,1]
Type Processing
Constraints
Avoid[ConflictOfInterestsWithAssociatedEditor]
Avoid[SurchargeAssociatedEditor],
Achieve[ListOfPotentialEvaluators],
Avoid[ConflictsWithEvaluator],
Maintain[CommittedEvaluator],
Avoid[SurchargeEvaluator],
Maintain[FeedbackOnPaper],
Maintain[ConfidentialityOfPapers],
Maintain[IntegrityOfPapers],
Maintain[ConfidentialityOfSubmission],
Maintain[IntegrityOfEvaluation],
Maintain[ConfidentialityOfSensibleDocument]
Composed of …
Uses
PeopleConnect *to interact with* (AutorAgent,
ChiefEditorAgent,
AssociatedEditorAgent,
EvaluatorAgent),
Conn1 *to interact with* Document,
Conn2 *to interact with* Paper,
…

Component Document [0,n]
Type Data
Constraints
Maintain[FeedbackOnPaper],
Maintain[IntegrityOfEvaluation]

Component Paper [0,n]
Type Data
Constraints Maintain[IntegrityOfPapers],

Component Conn1 [1,n]
Type Connector [1,n]
Constraints
Maintain[IntegrityOfEvaluation],
Maintain[ConfidentialityOfSensibleDocument]

…

Figure 7. A prescription for the paper selection process after step 3

We omitted the complete specification, but if we included it, it would be possible to notice that the components: ChiefEditor, Author, Knows, Holds, IsAuthorOf, Supervise, InChargeOf and Evaluates, which were potential sub-components at step 2, were removed from the prescription because they are not necessary to achieve the sub-goals for the system. This is due to the rationale that we took in prescribing the system. Different architects may use different rationales and produce different prescriptions.

At the third step (and at the optional fourth) we first consider the functional goals and than the non-functional ones. The goals of the latter type have a more complex effect on the system to achieve. In the most general case, apart from further constraining already existing components, they introduce new components and they transform the system's topology (i.e. they change the relationships among the system's components). Details on how the Preskriptor process manages non-functional requirements can be found in [8].

3.4 The fourth step

At this step of the prescription design process, the architectural prescription is further refined to make the system achieve goals that are not from the problem domain. These additional goals are typically introduced for a variety of reasons (for example architectural, economic, etc.).

These goals can be classified as follows: useful architectural properties, even though not required by the problem (such as reusability, evolvability, etc.), conformance to a particular architectural style, and compatibility goals (such as compatibility with a given platform or industry standard, or platform independency).

Examples of architectural goals are reusability, location transparency and dynamic reconfiguration. These goals can modify the prescription at the component level, at the sub-system level, or affect the whole system.

As practical experience has shown [8], architectural styles can be chosen as a particular solution to achieve some goals or to refine some components. For example, we can achieve the architectural goal of dynamic reconfiguration by making all the components adhere to the reconfigurable architectural style. By dynamic reconfiguration we mean that the application can evolve after it has been already deployed as demands change for new and different kinds of configuration. A reconfigurable architectural style is the following set of constraints: provide location independence; initialization must provide facilities for start, restart, rebuilding dynamic data, allocating resources, and initializing the component; finalization must provide facilities for preserving dynamic data, releasing resources, and terminating the component.

The last kind of goals that don't come from the problem domain are compatibility goals. They further constrain a prescription to take into account, already at this architectural design level, the need to assure the compatibility of the system with one or more industry standard(s) and/or platform(s). For example we may want to make a system CORBA or Linux compatible. This may be motivated by the need to assure compatibility with legacy systems, other vendors systems, available machines, or just for some marketing strategies.

Fig. 11 shows how step 4 interacts with the previous steps of the Preskriptor process.

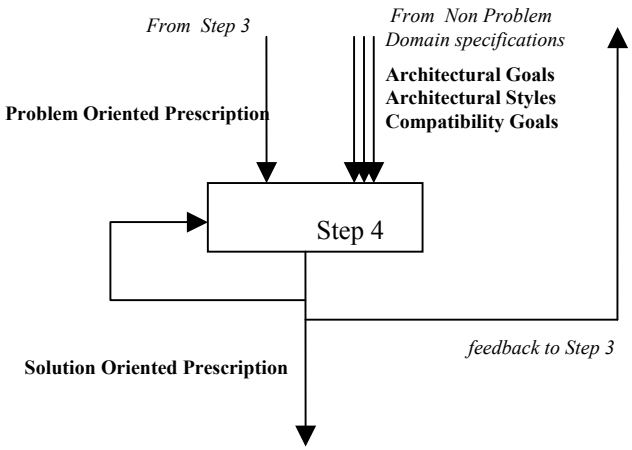

Figure 8: Step 4 of the Preskriptor process

As we can see, in general, the fourth step is iterated till we have achieved all of the non-domain goals. This step may also be iterated with step three. In that case, alternative problem domain goal refinements and/or components may be chosen to make the later prescription design steps possible or easier to perform.

It's important to distinguish between the artifact of the third step and the one of the fourth. The third step produces an artifact whose only constraints come from the problem domain, which can be reused with similar systems without over-constraining them. On the other hand after the fourth step we obtain a prescription that takes into account also constraints that we introduced for the particular product we are developing, such as the use of a particular architectural style or the compatibility with a certain industry standard. While the artifact of step four may be reused with other systems that we want to develop in a similar manner, we also want to be able to easily reuse a prescription in systems that are to be implemented with different non domain constraints, like with different architectural styles. For this reason we distinguish between the specification of the prescription after step 3., which we call Problem Oriented Prescription (POP), from the one after step 4, which we call Solution Oriented Prescription (SOP).

Given the Problem Oriented Prescription for the system and the non-domain driven goals, step 4 proceeds similarly to step 3. It takes as inputs a POP and the non problem domain goals, and gives a SOP as a result. In this step the non-domain goals are assigned as constraints to some POP components and/or the topology of the POP may be modified in order to achieve them (in this step we may reintroduce some of the KAOS components that we discarded at step three).

A Solution Oriented Prescription specification is similar to a POP specification, but it includes one or more of the following additional attributes: Architectural Goals, Architectural Styles and Compatibility Goals Specification. These new attributes are needed to keep track of the specifications of the goals, which don't come from the problem domain.

4. Conclusion

This paper presents an introduction to Preskriptor a method for transforming a requirements specification into an architectural prescription. Architectural prescriptions are a higher-level form of architectural specifications that interface more easily with requirements specifications and that do not include implementation oriented entities such as client-server which are often default components in architectural descriptions. We illustrated how to derive a prescription with a practical example. The key steps in the prescription specification process are: the selection of the right level of goal refinement, the choice of the potential components for the architecture, the assignment of the constraints to the potential components for the architecture and, often in the case of non-functional requirements, the modification of the architecture's topology.

Preskriptor is a systematic and rigorous process to make sure that none of the requirements are neglected, that no useless requirements and/or components are introduced and that the means for easily modifying the architecture are provided. The generality of our approach will allow the architects to choose their favorite ADL, or design specification, to describe at a lower level an architecture prescribed in Preskriptor.

The objectives for the future of our research are the extension of the methodology to take into account the most common non-functional requirements, the test of the methodology with case studies and empirical studies, and the development of a supporting tool.

5. References

[1] Jackson, M., "The world and the machine". *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington (USA), April 1995. Keynote speech

[2] Perry, D.E., Wolf, A. L., "Foundations for the Study of Software Architecture", *Software Engineering Notes*, ACM SIGSOFT, October 1992, pp. 40-52

[3] Parnas, D. L., "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, IEEE Computer Society, March 1976, pp. 1-9

[4] Brandozzi, M., and Perry, D. E., "Transforming Goal Oriented requirements specifications into Architectural Prescriptions", *Proceedings of STRAW '01, ICSE 2001*, Toronto, May 2001, 54-61

[5] Van Lamweerde, A., Darimont, R., and Massonet, P., "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems andLessons Learnt", *Proceedings RE'95 – 2nd IEEE Symposium on Requirements Engineering*, York, March 1995, pp. 194-203

[6] Van Lamweerde, A., Darimont, R., and Letier, E., "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Transactions on Software Engineering*, IEEE Computer Society, November 1998, pp. 908-925

[7] Cordier, C., and Van Lamweerde, A., "Analyse des constraintes desécurité puor la gestion électronique d'une revue scientifique", UniversiteCatholique de Louvain, 1997

[8] Brandozzi, M., and Perry, D., "Architectural Prescriptions for Dependable Systems", *Proceedings of WADS, ICSE 2002*, Orlando, May 2002

[9] Dewayne E. Perry. "A Product Line Architecture for a Network Product" *ARES III: Software Architectures for Product Families 2000*, Los Palmos, Gran Canaria, Spain, March 2000

6. Appendix

Preskriptor Specification: [Prescription's name]
(KAOS Specification: [Requirements specification's name])$^{?}$
Components:
(
Component **[Component's name]** ***([num1, num2])***$^{\&}$
Type {Processing | Data | Connector}
Constraints ([Constraint's name],)$^{+}$
(Composed of ([Component's name] *[num1, num2]*,)*)$^{?}$
(Extends [Component's name])$^{?}$
(Generalizes ([Component's name],)$^{+}$)$^{?}$
(Uses [Connector's name] *to interact with* ([Component's name],)$^{+}$)*
)$^{+}$

The terms between brackets denote the meaning of the identifier that will be at their place. "*" means that the preceding expression can be present 0 to an arbitrary number of times. "+" is the same except that it has to be present at least once. "?" means the expression can be present 0 or 1 time only. The new symbol "&" means that the expression is required only for the specification of the components at the first level of the components refinement tree.