Transforming Goal Oriented Requirement Specifications into Architecture Prescriptions

Manuel Brandozzi
Center for Advanced Experimental Software Engineering Research
The University of Texas at Austin
mbrandoz@ece.utexas.edu

Dewayne E. Perry
Center for Advanced Experimental Software Engineering Research
The University of Texas at Austin
perry@ece.utexas.edu

Abstract

In this paper we propose a new method to transform the requirements specification for a software system into an architectural specification for the system. In the introduction we illustrate the needs for this new method in the context of the software development process and we explain the concept of architecture prescription. Then, we give a brief overview of KAOS, the goal oriented requirements specification language we used as a starting point. We characterize the APL (Architecture Prescription Language) and show how to use it to derive an architecture prescription from the KAOS requirements. We then illustrate our technique with a practical example, namely the meeting-planning problem. Finally, we discuss related work and indicate future directions of our research.

1. Introduction

At present, there are many different software development processes used in industry. These processes may be more or less suited for the particular application being developed. A common characteristic of most of the development processes is that there is feedback from one phase to another one. Let's consider the earlier phases of a generic development process with feedback.

The process starts with the "requirements analysis and specification" phase. In this phase, the requirements engineer has to understand the user's needs and to document them, either formally or informally.

The second phase is the "architectural design" phase. In this phase, the system architect selects the architectural elements, their interactions and the constraints on these elements and interactions to achieve a basic framework to satisfy the requirements specified in the previous phase.

Then there is the design phase. During this phase the designer decides how to decompose the elements described in the architectural design into low level modules, which already existing components might be reused to implement these modules, or which algorithms and data structures should be used to implement the modules.

The later phases of the process include coding, testing, integration, delivery and maintenance. Each of the front-end phases can be viewed as the implementation of the previous one.

The process is iterative because, typically, either to make the implementation of a phase feasible, or more efficient, we return to previous phases, one or more times, and modify the relevant software artifacts of those phases. Coming back from one phase to a previous one constitutes considerable work and time overhead because it's generally very difficult to understand what has to be modified in the previous phase and these modifications may have side effects. So, there is a need to minimize the number of iterations, or to make them easier to perform and easier to identify their side effects whenever they are necessary. Iterations in the process happen also when it is discovered that the system doesn't do what the user really wanted. So, we feel we can improve the process by using better methods to specify the requirements and by using rigorous techniques to pass from each phase of the development process to the next one.

There are further advantages of passing from a phase to the next one using formal techniques. By doing this, we achieve reusability of part of the artifact of each phase. The earlier the phase the artifact belongs to, the higher is the gain obtained. Let's suppose, for instance, that we have to develop a new system for which only a small part of the requirements differ from those of a system already developed. With our approach that maps the requirements to the components derived from them, we know exactly what has to be changed in the architecture of the system already developed in order to obtain an architecture of the new system. The same

applies when some of the requirements of a system are changed, for whatever reason, during or after the development of the system.

As experience shows, in traditional approaches the modification of requirements might have very subtle effects on the architecture and a brand new architecture may be needed each time requirements are modified if we are to achieve a reliable system. The method we introduce in this paper, by providing requirements to architecture traceability, enables us to reuse parts of the architecture, and hence to reuse all the derived artifacts that implement the architectural components. It enables a development team to save both time and resources.

Our work has been focused on finding a method for the first of the transitions from one phase to the next: the transition going from the requirements specification to the architectural design, i.e. the one that has the highest leverage. Traditionally, this transition has been one of the most difficult aspects of software engineering. The primary problem in software development is transforming *what* we want the system to do into a basic framework for *how* to do it. Our method takes as input goal oriented requirement specifications and returns as output an architecture prescription. An architecture prescription is an alternative way to specify an architecture. We chose goal oriented specifications because we think they are, among all the kinds of requirements specifications, those more near to the way human thinks and are easy to understand by all the stakeholders. Another reason is that they are particularly suitable to be transformed into an architecture prescription. In the next section we'll give a brief description of KAOS, the goal oriented specification language that we used in our example that has been first introduced by Axel van Lamsweerde et al. [1].

Let's now explain what we mean by an architecture prescription, a concept introduced by Dewayne E. Perry and Alexander L. Wolf [3]. An architecture prescription lays out the space for the system structure by restricting the architectural elements (processes, data, connectors), their relationships (interactions) and constraints that can be use to implement the system. The main advantages of an architecture prescription over a typical architecture description are that it can be expressed in the problem domain language and it's often less complete, and hence less constraining with respect to the remaining design of the system. An architectural prescription concentrates on the most important and critical aspects of the architecture and these constraints are most naturally expressed in terms of the problem space (or business domain, the domain of the problem). An architecture description, on the other hand is a complete description of the elements and how they interface with each other and tends to be defined in terms of the solution space rather than the problem space (or in terms of components such as GUIs, Middleware, Databases, etc, that are used to implement the system). Since an architecture prescription is expressed in the domain language, it makes it easier to create a mean of transforming requirement specifications into architectural specifications. The two kinds of specifications can make use of a common vocabulary to relate the requirements' goals to the architectural constraints.

The purpose of our work is twofold: to propose architecture prescriptions as a way to specify the architectures of software systems, and to design a technique to transform the requirements specifications into prescriptive specifications.

The rest of the paper is structured as follows: in section 2 we give an overview of KAOS, in section 3 we show how to derive from a KAOS specification the architecture prescription whose architectural elements, and the way these elements interact, are defined via application specific constraints. In section 4 we'll give a practical example of the method using the meeting-planning problem and, finally, in section 5 we will summarize the contribution of our work and illustrate its future directions.

2. Overview of the KAOS Specification Language

KAOS is a goal oriented requirements specification language [1]. Its ontology is Composed of:

- *Objects* - they can be:
  - *Agents*: active objects
  - *Entities*: passive objects
  - *Events*: instantaneous objects
  - *Relationships*: depend on other objects

- *Operations*: they are performed by an agent and change the state of one or more objects. They are characterized by pre-, post- and trigger-conditions.

- *Goal*: it's an objective for the system. In general, a goal can be AND/OR refined till we obtain a set of goals achievable by some agents by performing operations on some objects. The refinement process generates a refinement tree.

- *Requisites, requirements and assumptions*: the leaves obtained in the goal refinement tree are called requisites. The requisites that are assigned to the software system are called requirements; those assigned to the interacting environment are called assumptions.

How are requirements specified? 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 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 [2]. It's up to the requirements engineer to choose the best refinement tree. A refinement tree could be modified afterwards in case there are problems implementing the artifacts of a latter phase of the development process.

In exhibit 1. there is an example of a goal specified using KAOS.

Goal *Achieve*[MeetingRequestSatisfied]
InstanceOf SatisfactionGoal
Concerns Meeting, Initiator, Participant
ReducedTo SchedulerAvailable,
ParticipantsConstraintsKnown,
MeetingPlanned,
ParticipantsNotified
InformalDef Every meeting request should be satisfied within some deadline associated with the request. Satisfying a request means proposing some best meeting date/location to the intended participants that fit their constraints, or notifying them that no solution can be found with those constraints.

Exhibit 1. example of a goal specification in KAOS

The *Goal* keyword 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* traces into which sub-goals the goal is resolved. Finally, there is informal definition of the goal followed by an optional formal definition. *FormalDef* is the optional attribute; it contains a formal definition of the goal that can be expressed in any formal notation.

3. From Requirements to Architecture

3.1 From KAOS entities to APL entities

How is it possible to transform a KAOS requirements specification into an architecture prescription for the software system? Exhibit 2. shows the correspondence we found between KAOS entities that refer to a subset of the system specification and the Architecture Prescription Language (APL) entities that describe the constraints on the software architecture. The subset of the overall system specification considered is the subset concerning the software system specification.

KAOS entities	**APL entities**
•Agent	•Process component / Connector component
•Event	•Event
•Entity	•Data component
•Relationship	•Data component / Relationship among components
•Goal	•Constraint on the system or on a subset of the system •One or more additional processing, data or connector components

Exhibit 2. Mapping KAOS entities to APL entities

Each *object* in the requirements generally corresponds to a *component* in the architecture. More specifically, an *agent object*, an active object, corresponds to either a *process* or a *connector*. By definition, a process (thread, task) is an active component. What might not be immediately apparent is that also a connector can be an active component. An example of this type of connector is a software firewall. A software firewall is an active entity that checks whether the processes that want to interact satisfy some conditions or not, and allows or denies the interaction among them accordingly.

The *events* relevant to the architecture of the system are those either internal to the software system or those in the environment that have to be taken into account by the software system. The receiving of a message by a process is an example of internal event. The triggering of an interrupt by a sensor is an example of external event. An event is generally associated to a connector.

An *entity*, or passive object, corresponds to a *data* element, which has a state that can be modified by active objects. For example, the speed of a train is a variable (entity) that can be modified by a controller (agent).

A *relation* corresponds to another type of *data* element that links two or more other objects and that can have additional attributes. An example of relation data is a data structure whose attributes are the type of train, its

current speed and its maximum speed (additional attribute).

A *goal* is a constraint on one or more of the components of a software system. Additional components may be derived to satisfy a non-functional goal. An example of a constraint deriving from a goal is that a component of the software system of an ATM has to check if the password typed by the user matches the password associated in the system to the ATM card inserted.

3.2 The Architecture Prescription Language

Appendix A shows an abstract example of the refinement tree for the goals (on the left), and of the refinement tree for the corresponding architecture prescription components (on the right). As the example shows, the trees don't have the same shape. It would be a pure coincidence if they did have it.

The goal refinement tree is obtained as we explained in section 2. All the refinements are pure "and" apart from the refinement of goal G1. G1 is obtained by achieving requirement R1.1 and either requirement R1.2 or goal G1.1 (the arch between R1.2 and G1.1 denotes an "or" refinement). The sub-goals/requirements refining goal Gi are denoted as Gi.j, with j varying from 1 to the number of sub-goals/requirements. We use an analogous notation for the subcomponents.

In the component refinement tree, the root component C is the software system itself. The software system is viewed as a component of the bigger system that may include hardware devices, mechanical devices and human operators. We want to note here that also for these other kinds of systems we could design an architecture prescription language. Ours, anyway, is tailored to the software sub-system. The first refinement of C is obtained by considering the components directly derived by the KAOS specification by using the methodology we explained in section 3.1. Note that we may provide further refinements or even redo existing refinements due to non-functional requirements such as performance and reliability or from reusability considerations.

Exhibit 3. shows how the APL describes all the attributes of the components in the refinement tree. Please note that the Composed of relationship is the only one that can be deduced directly from the tree.

Component C:
KAOS spec.: S
Type: Software System
Constraints: R1.1, (R1.2 or (R1.3.1, R1.3.2)), R2.1, R3.1, R3.2
Composed of: C1, C2, C3, C4
Uses: /

Component C1:
KAOS spec.: S
Type: Processing
Constraints: R1.1, R3.1a
Composed of: C1.1, C1.2, C1.3
Uses: C2 *to interact with* {C3}

Component C2:
KAOS spec.: S
Type: Connecting
Constraints: R3.1b
Composed of: C2.1, C2.2
Events: E1, E2, E3
Uses: /

Component C3:
KAOS spec.: S
Type: Data
Constraints: R1.2, R2.1
Composed of:/
Uses: C2 *to interact with* {C1, C4}

Component C4:
KAOS spec.: S
Type: Processing
Constraints: R1.1, R3.2
Composed of: C4.1, C4.2
Uses: C2 *to interact with* {C3}

Component C1.1:
KAOS spec.: S
Type: Processing
Constraints: R1.1
Composed of:/
Uses:/

Component C1.2:
KAOS spec.: S
Type: Connector
Constraints: R3.1a
Composed of:/
Uses: C2 *to interact with* {C3}

Component C1.3:
KAOS spec.: S
Type: Processing
Constraints: R1.1
Composed of:/
Uses:/

Component C2.1:
KAOS spec.: S
Type: Data
Constraints: R3.1b.1

Composed of:/
Events: E1, E2
Uses:/

Component C2.2:
KAOS spec.: S
Type: Processing
Constraints: R3.1b.2
Composed of:/
Events: E2, E3
Uses:/

Component C4.1:
KAOS spec.: S
Type: Processing
Constraints: R1.1
Composed of: /
Uses: C2 *to interact with* {C3}

Component C4.1:
KAOS spec.: S
Type: Data
Constraints: R3.2
Composed of: /
Uses: C2 *to interact with* {C3}

Exhibit 3. APL prescriptions

The attribute *KAOS spec.* denotes the specification from which the component is derived. In our example we called this specification S.

Type specifies the type of the component. The possible types for components are: *Software System*, *Processing*, *Connecting* and *Data*.

Constraints is the most important attribute of a component. It denotes which requirements the component satisfies. For example, the root component C, i.e. the software system, must achieve all the goals. The subcomponents in the first layer of the tree, instead, have to satisfy only a subset of the system requirements. The union of the requirements achieved by the leaves components is the complete set of requirements.

Also, a component may be only contributing in achieving a goal without being able to achieve it alone. This may happen in case of non-functional requirement such as security. When a component cannot achieve a requirement only by itself, we represent it in our prescription language by appending a different lower case letter to the name of that requirement in each of the components involved in achieving it. In our example, this happens with C1 and C2. In order to achieve goal R3.1, goals R3.1a and R3.1b have to be achieved by C1 and C2 respectively.

The same requirement can be achieved by more than one software component. One reason for such a redundancy might be a reliability goal; another reason might be that the achievement of a goal may best be done cooperatively. In refinements successive to the first one (in which all the components are directly derived from the requirements specification) the constraints themselves can be further refined in order to better allocate them to different subcomponents. In the next paragraph we'll explain the reasons for such subcomponents. So, it may happen what we show in our abstract example. In our example C2.1, a subcomponent of C2, whose constraints are requirements R1 and R3.1.b, has as constraint requirement R3.1b.1 (R3.1b.1 is one of the two sub-requirements that and-refine R3.1b). The other subcomponent of C2, C2.2 has R3b.2 as constraint.

Composed of identifies the subcomponents that implement the component. The subcomponents of the root component are obtained directly from the KAOS specification. The subcomponents in the next layers of the components refinement tree are designed by the software architect in order to make the software system achieve other desirable characteristics, such as better performance or greater reusability of the components (even across different domain). For example, a component directly derived from the KAOS specification might be too big; it could have too many requirements as constraints. If the component implements many requirements many users/software components might use it. This would lower the system performance. Also, reducing the constraints on a component will make it easier to modify the component in case one of the requirements is removed or changes during the software development process. To achieve this purpose the component could be split into many subcomponents that have fewer requirements or even only a part of a requirement. As we said in the previous paragraph, some requirements might be further refined after the specification phase, even though they are already directly achievable by some agents. Having to satisfy only a sub-requirement may make the subcomponent more reusable. For example, a requirement on a component for an intelligent house software system might have to take into account both inputs from a smoke detector and a heat sensor to detect a fire. Even though the requirement can be directly achieved by a single component, to make the fire manager components more portable (to a system that has a smoke sensor only, for example) as well as better maintainable, we can split the requirement into smoke detection and heat detection sub-requirements and assign them to different components.

The attribute *Events*, generally assigned to connector components, indicates the events the component has to handle.

The last attribute, *Uses*, indicates what are the components used by the component. Since interactions always happen through a connector, the *Uses* attribute has

the optional keyword *to interact with* that indicates which components the component interacts with using that connector.

4. An Architecture Prescription for the Meeting-Planning Problem

Now, we will know show how to obtain an architecture prescription in practice. For this purpose we will consider the meeting-planning problem. At the highest abstraction level there are two goals that every meeting planner has to achieve. They are (in KAOS notation):

Achieve[MeetingRequestSatisfied]
Maximize[NumberOfParticipants]

We already showed the specification of the first goal (exhibit 1.). From this goal specification we obtain three agents one of which is software: Scheduler. From the sub-goal Achieve[SchedulerAvailable] we deduce that the root component of the architecture prescription must be parent also of other two components: SchedulerManager and MConnector. Scheduler Manager finds an available scheduler, communicating to the existing schedulers by MConnector, and in case no Scheduler is available it builds a new one. We call the root component MeetingWizard.

The second goal translates in an additional constraint on Scheduler.

Without going into the details of the KAOS specification for the meeting planner, some of which are in [1] and [2], in exhibit 6 we show some of the components of the meeting planner architecture prescription that illustrate many of the characteristics we discussed before.

Component MeetingWizard:
KAOS spec.: MeetingPlanner
Type: Software System
Constraints: {the complete set of requirements}
Composed of: Scheduler, SchedulerManager, MConnector
Uses:/

Component Scheduler:
KAOS spec.: MeetingPlanner
Type: Processing
Constraints: {the complete set of requirements} \ Achieve[SchedulerAvailable]
Composed of: PlanningEngine, ParticipantClient, MeetingInitiatorClient, ResourcesAvailableRepository, SecureConnector1, SecureConnector2
Uses:/

Component SchedulerManager:
KAOS spec.: MeetingPlanner
Type: Processing
Constraints: Achieve[SchedulerAvailable]
Composed of:/
Uses: Scheduler, MConnector *to interact with* {Scheduler}

Component PlanningEngine:
KAOS spec.: MeetingPlanner
Type: Processing
Constraints: {subset of the set of requirements}
Composed of: Planner, Optimizer
Uses: SecureConnector *to interact with* {ParticipantClient, MeetingInitiatorClient}

Exhibit 6. APL sample prescription for the meeting planner

5. Conclusion

In this paper we have illustrated the advantages of formal techniques to go from a phase of the software development process to its next one. We focused on what we consider the most important of these transitions: the one from requirements to architecture. To make a formal transition between these two phases easier we have introduced an architecture prescription language (APL), that specifies the structure of the software system and its components in the language of the application domain. This higher-level architecture specification can be then easily translated, if necessary, in an architecture description, in the solution domain. We took advantage of the characteristics of KAOS as a requirements specification language.

Other researchers in the past have tried to find techniques to pass from requirements to architecture. Nenad Medvidovic et al., in [4], developed a technique to pass from requirements specified in WinWin to an architectural model for the system. Their technique, while providing a framework to pass from requirements to architecture, is not formal and still leaves a lot of choices to the architects. This due in part the big gap between the requirements specification, specified in the problem domain language, and architectural design, described in the solution domain language. Other researchers have designed object-oriented techniques to pass from requirements to architecture. These techniques, though,

are still very informal with little guidance for the architect on to decompose the architecture into classes and which attributes and methods to assign to those classes. Furthermore, this approach is tailored to an object oriented design.

Our goal is to design a formal technique to pass from the requirements to an architecture prescription that can be refined afterwards. The formality is necessary to make it sure that none of the requirements are neglected, and that we don't introduce any useless component or constraint. The generality of our approach allows the architects to choose their favorite ADL (architecture description language) to describe an architecture prescribed in APL.

Future areas of our work will include: define and perform experiments that tests our method, further research or even redefine the components of the APL to achieve non functional properties such as better performance and reusability, and build supporting tools that take the requirements for a software system and some other parameters and transform them into an architecture prescription for the system.

6. References:

[1] Anne Dardenne, Axel van Lamweerde and Stephen Fickas, "Goal-directed Requirements Acquisition", Science of Computer Programming, Vol.20, 1993, pp. 3-50

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

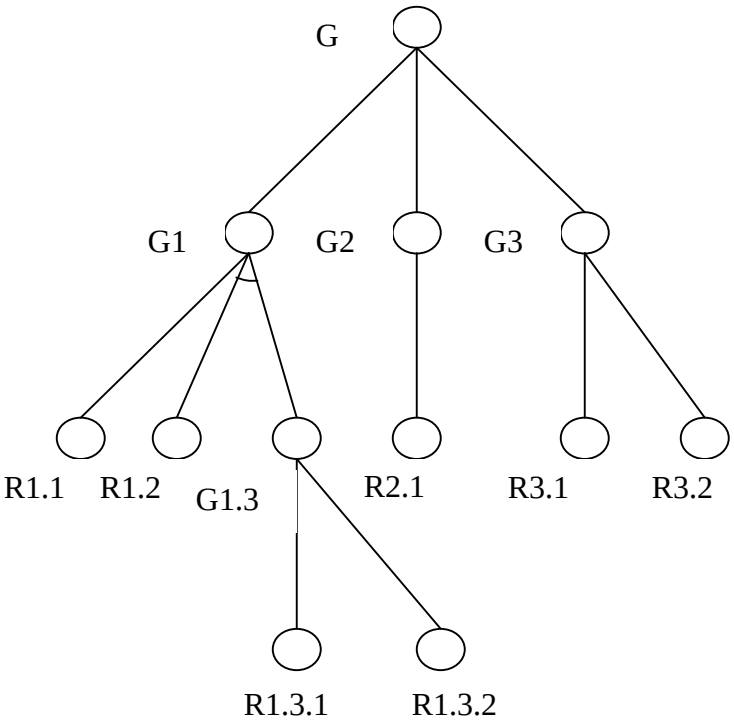
[3] Dewayne E. Perry, Alexander L. Wolf, "Foundations for the Study of Software Architecture", Software Engineering Notes, ACM SIGSOFT, October 1992, pp. 40-52

[4] Nenad Medvidovic, Paul Gruenbacher, Alexander F. Egyed, and Barry W. Boehm, "Bridging Models across the Software Lifecycle", Technical Report USC-CSE-2000-521, University of Southern California

[5] Axel van Lamsweerde, Robert Darimont, and Philippe Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt", Proceedings RE'95 – 2nd IEEE Symposium on Requirements Engineering, York, March 1995, pp. 194-203

Appendix A.

Goal Refinement Tree



Component Refinement Tree