Copyright

by

Manuel Brandozzi

2001

From Goal Oriented Requirements Specifications to Architectural Prescriptions

by

Manuel Brandozzi

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2001

From Goal Oriented Requirements Specifications to Architectural Prescriptions

Approved by
Supervising Committee:

_____

_____

Dedication

To My Family

Acknowledgements

I'd like to thank my supervisor, Dr. Dewayne Perry, for his invaluable guidance on my work. I also would like to thank Dr. Carlo Ghezzi for his insightful suggestions.

December 2001

Abstract

From Goal Oriented Requirements Specifications to Architectural Prescriptions

Manuel Brandozzi, M.S.E.

The University of Texas at Austin, 2001

Supervisor: Dewayne E. Perry

In the present thesis we propose a new method to design a high level architecture of a software system that guarantees the satisfaction of its requirements. We give an overview of the concepts of *goal oriented requirements specifications* and of *architectural prescriptions*; we summarize the characteristics of *KAOS*, the goal oriented requirements specification language that is used by our process, and we introduce *Preskriptor*, our Prescription Specification Language (APL); we illustrate our methodology for transforming requirements to an architectural prescription and provide practical examples; finally, we discuss related work and indicate further directions of our research.

Table of Contents

List of Figures

Chapter 1: Introduction

Traditionally, the most difficult transition of the development process for a non-trivial software system has been the one from the requirements for the system to its design. This step involves going from the problem's domain to the domain of its solution [Jack 95]. One of the factors that make 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 not a single solution to the problem, rather many possible ones.

Requirements specifications can be viewed as a contract between the customer and the software developers and they should be easy to understand by domain experts and users, as well by software architects and engineers.

Let's now introduce our methodology and insert it in the context of the software development process. At present, there are many different software development processes that are used in industry and that have been studied in academia. A common characteristic of most of these development processes is the feedback from one phase to another one as proposed in the spiral model [Boehm 88].

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 customer's and/or user's needs and document them, in a requirements specification language.

The second phase is the "architectural design" phase. In this phase, the system architect selects the architectural elements, their interactions and the constraints on the elements and interactions that provide a basic framework to satisfy the requirements specified in the previous phase. The low level design phase follows. During this phase the designer decides how to decompose the elements described in the architectural design into low level modules, i.e. modules that include the detailed specification of data and processes used and exported.

Each of the front-end phases can be viewed as the implementation or refinement of the previous one. Iterations in the process may happen because the developers discovered that the system doesn't do what the customer/user really wanted. So, we feel that we can improve the process by introducing a technique that provides guidance in going from requirements to architecture.

The method that we introduce takes as input the system's requirement specifications (expressed in a goal oriented language), and provides as output an architecture specification (expressed in an architectural prescription language). Architecture prescriptions [Perry 92] are specifications of the system's basic components and topology, and of the constraints associated with its components and interactions. Furthermore, the constraints are expressed in terms of the problem domain as opposed to the solution domain. For example, a problem domain constraint on a component may be to be able to handle electronic mail or streaming video. So, an architectural prescription specifies higher level components such as email managers and streaming video players rather than GUI's and databases. A by product of our approach is requirements to

architecture traceability, which enables us to reuse parts of an architecture, and hence to reuse all the derived artifacts that implement it. This translates into a saving of both time and resources for the development team.

The purpose of our work is twofold: to advocate the use of architecture prescriptions in the specification of the architectures of software systems, and to introduce a process that performs the step from the requirements specifications to architecture prescriptions.

Chapter 2: Overview of Goal Oriented Requirements Specifications and KAOS

We chose goal oriented specifications because we think that they are, among all the kinds of requirements specifications, those that are closer to the way humans think and hence easier to understand by all the stakeholders in the development process. Another reason is that they can be refined from higher level goals to lower level ones. A refinement of a goal is constituted by a set of goals that once achieved imply the achievement of the original goal. A refinement is in general composed by a conjunction and disjunction of nodes (called AND / OR refinements respectively). For example we can achieve the goal "go home" by achieving the goals "buy bus ticket", "catch a bus" and "get off at home" or by achieving the goals "take the car", "drive home" and "get off". While in this example the goals have to be achieved in the specified order, it's not always the case. We can vary the constraining level of an architecture prescription by considering different levels of refinement of the requirements.

In the remainder of the chapter we'll give a description of the main characteristics of KAOS, the goal oriented specification language, introduced by A. van Lamsweerde [Lam 95], that we used in our methodology.

KAOS' ontology is composed of *objects*, *operations* and *goals*. *Objects* can be agents (active objects), entities (passive objects), events (instantaneous objects), and relationships (objects depending on other objects). *Operations* are performed by an agent, and change the state of one or more other 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 can be denoted requirements; those assigned to the interacting environment can be called assumptions.

Let's briefly see now 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 [Lam 98]. 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 design process.

Figure 1. is an example of a goal declaration in KAOS that is taken from the example that we will use extensively in the following chapters.

Goal *Maintain*[ConfidentialityOfSubmissions]
InstanceOf SecurityGoal
Concerns DocumentCopy, Knows, People
ReducedTo
ConfidentialityOfSubmissionDocument
ConfidentialityOfIndirectSubmission
InformalDef A submission must remain confidential. An article 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. Finally, there is *InformalDef*: the informal definition of the goal. *FormalDef* is an optional attribute; it contains a formal definition of the goal (it can be expressed in any formal notation such as first order logic).

Chapter 3: Architectural Prescriptions

An architecture prescription lays out the space for the system structure by selecting the architectural elements (processes, data, and connectors), their relationships (interactions) and constraints. In a prescription, the most important characterization of the components is given by the goals they are responsible for (i.e., their constraints). Components are further characterized by their type: process, data or connector. The processing components are those the provide the transformation on the data components. The data components contain the information to be used and transformed. The connector components, which may be either implemented by data components, processing components or by a combination of both, are the glue that holds all the pieces of the system together. The interactions of the components among each other, together with the restriction of their possible number of instances, characterize the topology of the system.

The main advantages of an architecture prescription over a typical architecture description are that it is expressed in the problem domain language, it is often less complete, and hence less constraining with respect to the next phases of system design. An architecture description, on the other hand is, generally, 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 problem domain language, it's also easier to create a method to design it starting from the requirements specifications. The two kinds of specifications can make use of a common logic and vocabulary mapping the requirements specifications' goals to the architectural constraints. In the following chapters we will show how we perform the step from goal-oriented requirements. Also, being at a higher level of abstraction, prescriptions can more easily be reused and they enable more creative designs.

Let's consider, for example, a distributed system. An architecture description language (ADL) may include elements such as clients and servers. So, it will be likely that the architect writing a specification in that ADL will use client and server components also when the best way to solve the problem was another one (for example a multi-peer architecture). Then, the designer will be constrained by the architecture to a low-level design that adopts a client-server solution. Since an APL specifies the system at a higher level of abstraction, it would permit the designer to choose a better (possibly more innovative) solution at the low-level design and even to implement different choices for different members of the same software family.

Chapter 4: From KAOS Entities to APL entities

How is it possible to transform a KAOS requirements specification into an architecture prescription for the software system? Figure 2. shows what are the effects, in terms of topological transformations and constraints, of the KAOS entities and how they relate the specification to the system's architecture prescription. Note that, in general, a requirements specification considers a system of which the software system is only a part. From here on, by requirements specification we will mean the subset of the specification that concerns the software system, unless otherwise stated.

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

Figure 2: Mapping KAOS entities to an APL entities.

The "-" symbol means no effects on the architecture. Note that only the *Goal* entities are guaranteed to affect the prescription. Each object in the

requirements may generate a corresponding a component in the prescription; we'll see in which case this happens in an example in chapter 6. An *agent* object, i.e. an active object, may generate 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 sending and receiving of messages by processes are an example of internal events. The triggering of an interrupt by a sensor is an example of external event. An event is not associated to any architectural component but is has to be taken into account by the prescription through the goals that depend on it.

An *entity*, or passive object, may correspond 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 *relationship* may correspond to a *data* element too. An example of relation data is a data structure that associates a trains and their current speed.

A *goal* corresponds to 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 on a particular component deriving

from a goal is that a particular component of the software of an ATM has to check whether the password typed by a user matches the password associated in the system to the ATM card that he/she inserted.

Chapter 5: Preskriptor - an Architecture Prescription Language

Now we'll introduce Preskriptor, the APL we use in our methodology. The software system that we'll use as an example, is a system that helps in the paper selection process for a scientific magazine (or conference). We will hereafter denote it as "ScientificPaperSelector".

ScientificPaperSelector is co-responsible for the root goal "Maintain[QualityOfTheScientificMagazine]" together with the system it interacts with, (i.e., the people involved in the process). *ScientificPaperSelector* performs different functions that can be automated and it interacts with the system composed of people. Its purpose is to speed up the paper selction process and to improve its confidentiality. For the KAOS requirements specification of the system, we consider the specification in [Cordier 97].

Fig. 3. shows the refinement tree for its requirements specification, and Fig. 4 is the refinement tree for the corresponding architecture prescription. A refinement of a component of a prescription is a set of sub-components that are used to implement it. A refinement tree shows the refinements of all the goals or components for the system. If there is an arch connecting two outgoing edges from the same goal node, the node it's OR refined by the goals; if there is no such arch, it's AND refined. A component is refined by the nodes belonging to its outgoing edges.

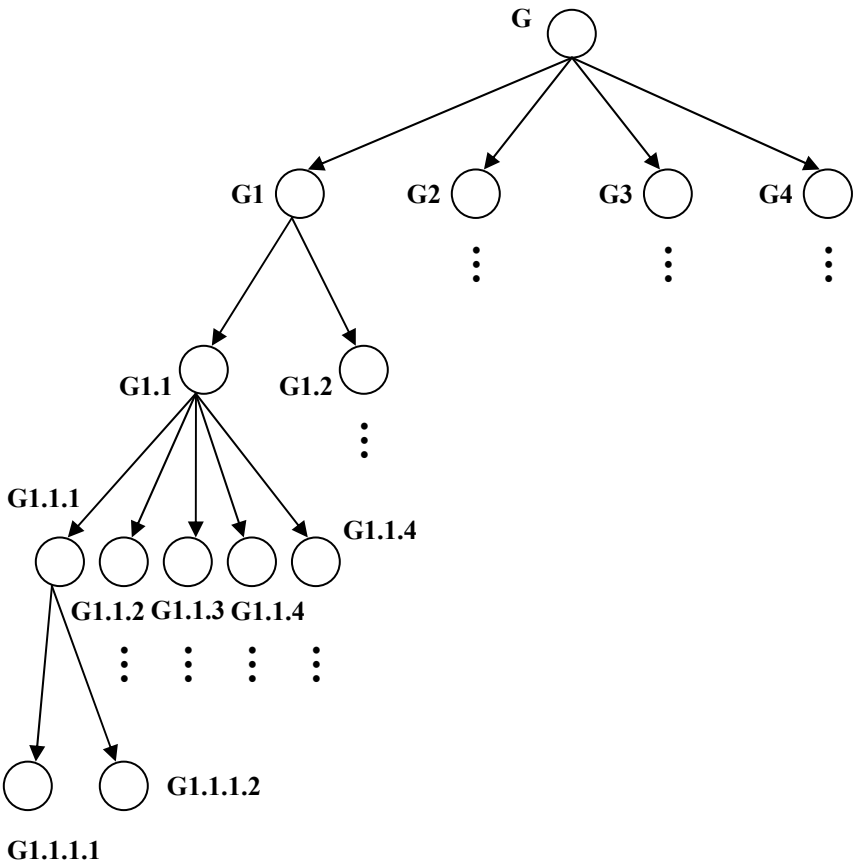

Figure 3: Goal refinement tree for the paper selection system, the refinements are all AND.

Let's have an overview of the goal refinement tree in figure 3.

Goal G is the root goal, the fundamental goal for the system:

G:Maintain[QualityOfTheScientificMagazine]".

G is AND refined by the goals:

G1: Maintain[QualityOfPublishedArticles],

G2: Maintain[OriginalityOfSubmission],

G3: Maintain[QualityOfPrint],

G4: Achieve[EnoughQuantityOfPublishedArticles].

Goal G1 is AND refined by:

G1.1: Maintain[QualityOfEditorialDecisions],

G1.2: Maintain[PertinenceOfPublishedArticles].

Continuing in this manner, we refine all the other goals till, for each goal, we get the level of refinement we want.

As the example shows, the two goals and components trees are not similar. They are built using different processes. The goal refinement tree is obtained in the requirements specification process. The component refinement tree is obtained during the prescription design process that we'll introduce in following chapter. It consists in performing the architectural transformation that the requirements specification components may generate as figure 2 shows.

All the refinements in goal tree of figure 3. are pure AND. The sub-goals refining goal Gi are denoted as Gi.j, with j varying from one to the number of sub-goals/requirements. We use an analogous notation for the subcomponents in the other tree.

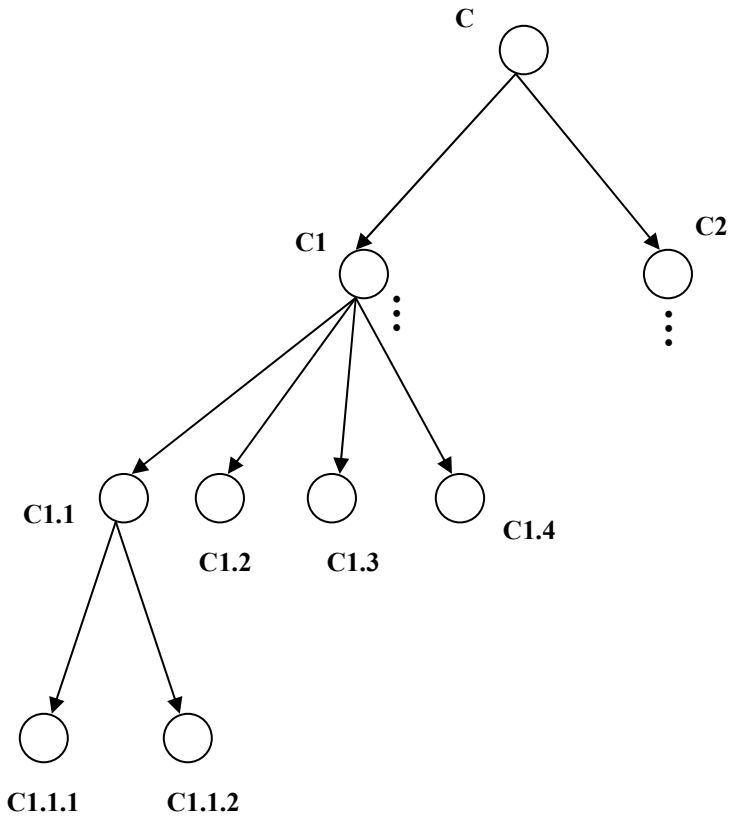

Figure 4: Component refinement tree for the paper selection system.

In the component refinement tree, the root component C is the software system itself: *ScientificPaperSelector*. The software system is viewed as a component of the bigger system that may include hardware devices, mechanical devices and human operators. As we will see in next chapter, the first refinement is performed by considering the root goal and the systems the software system has to interact with. Component C is refined by:

C1: SelectionManager,

C2: PeopleConnect

The second refinement layer is obtained by considering components derived by the KAOS specification. At this layer component C1 is refined by several components:

C1.1: SelectionManagerEngine

C1.2: Document

C1.3: People

C1.4: Evaluator

…

As the tree shows, we may have further refinement layers in this tree, to achieve additional non requirements such as performance, reliability or reusability. For example, as we will see in chapter 7, SelectionManagerEngine can be refined by PeopleSelectionManager and SelectionProcessManager, (components C1.1.1 and C1.1.2) to improve reusability.

Figure 5. shows how Preskriptor, the architectural prescription language, specifies the components that we see in the component refinement tree. The appendix contains Preskriptor's grammar. Please note that the *Composed of* attribute is the only one that can be deduced directly from the components refinement tree.

Preskriptor Specification: ScientificPaperSelector
Problem Goals Specification: PaperSelectionProcess (KAOS)
Components:

Component SelectionManager *[1,1]*

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

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

Component SelectionManagerEngine *[1,1]*
Type Processing
Constraints
Avoid[ConflictOfInterestesWithAssociatedEditor],
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 People *[0,n]*
Type Data
Constraints
Avoid[ConflictOfInterestesWithAssociatedEditor],
Avoid[ConflictsWithEvaluator],
Achieve[ListOfPotentialEvaluators]

Component Evaluator *[0,n]*
Type Data
Constraints
Avoid[SurchargeEvaluator],
Maintain[CommitedEvaluator]

Component PeopleSelectionManager *[1,1]*
Type Processing
Constraints
Avoid[ConflictOfInterestesWithAssociatedEditor],
Avoid[SurchargeAssociatedEditor],
Achieve[ListOfPotentialEvaluators],
Avoid[ConflictsWithEvaluator],
Maintain[CommitedEvaluator],
Avoid[SurchargeEvaluator],
Composed of …
Uses PeopleConnect *to interact with* (ChiefEditorAgent,
AssociatedEditorAgent, EvaluatorAgent)

Component SelectionProcessManager *[1,1]*
Type Processing
Constraints
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,

Figure 5: Example of components' specification.

As we can see, an architectural specification in Preskriptor has to start with the declaration of the name of the system, *ScientificPaperSelector* in the example. It follows the declaration of the problem goal oriented requirements specification, *Problem Goals Specification*, from which the prescription is derived. As we will see later on, this doesn't have to be necessarily expressed in KAOS. For this reason, the name of the specification language used is indicated between parentheses. The prescription has only a requirements specification as attribute because even when it derives from more than a specification, it's better to merge the specifications first and then design the prescription. In fact it's easier, if there are any conflicts between goals in different specifications to solve them will be solved at the requirements specification phase. This means that all the components of a prescription will derive from the same goal oriented specification, that is, in general, the union of more specifications. In our example we called this specification, which is in KAOS, *PaperSelectionProcess*.

Type denotes the type of the component. Again, the possible types of components are: Processing, Connecting and Data.

Constraints is the most important attribute of a component. It denotes which requirements that the component is responsible for. For example, the root component C, i.e. the software system, must achieve all the goals. Its subcomponents, instead, are responsible for only a subset of the system

requirements. The union of the requirements achieved by the leaves components has to be the complete set of requirements.

A component may be only contributing in achieving a goal without being able to achieve it alone. This may happen, for example, in the case of non-functional requirements like security. The same requirement can be achieved by more than one software component. Such a redundancy may come, for example, from a reliability goal.

Composed of identifies the subcomponents that implement the component in the next refinement layer. At the first layer of abstraction we have to write next to the name of a component its possible number of instances in the system. At the other layers this is optional because this information will be contained anyway in the *Composed of* field of its super-components. For example, *[1,n]* means that the component can have any number of instances from 1 to an arbitrary number. *[1,1]* means there has to be and there can only be a single instance of the component.

The last attribute, *Uses*, indicates what 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* that indicates which components the component interacts with using that connector. The symbol "/" means no attribute and, for now, we will omit the fields whose value is none.

Another attribute for a component, whose value in the example was always "/", is *Specializes*. This attribute is a syntactic shortcut for the specification of components that only have some additional constraints with

respect to other components. It is particularly useful for the prescription of software families. *Specializes* does not preclude the presence *Composed of*; the component including the *Specializes* attribute is a modified version of the component(s) that the attribute identifies.

Architecting practical experience tells us that it would be useful to have also a *Generalizes* attribute, to make it easier to design a prescription for a software family by generating it from initially independent applications (bottom-up). *Generalizes* means that the component takes all and only the common characteristics of the components identified by the attribute.

Chapter 6: From Domain Requirements to Architecture Prescriptions

Figure 6. is a schematic representation of the process we propose to derive an architecture prescription, such as the one we discussed in chapter 5., from a requirements specification. By using a goal oriented requirements specification as a starting point, we can gradually increase the degree of constraint of the solution by considering the goals that refine those used previously.

If we take the root of the tree, the resulting prescription may enable new, innovative solutions to the problem, but 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 further refinements of the prescription to achieve qualities as performance, reusability, etc.) may produce a specification that constrains too much of the lower level designs. As D. Parnas once noted, if in order to design washing machines we used all the requirements coming from how we washed the clothes by hand through the centuries, we would never have been able to achieve the very successful rotary washing machines of our days.

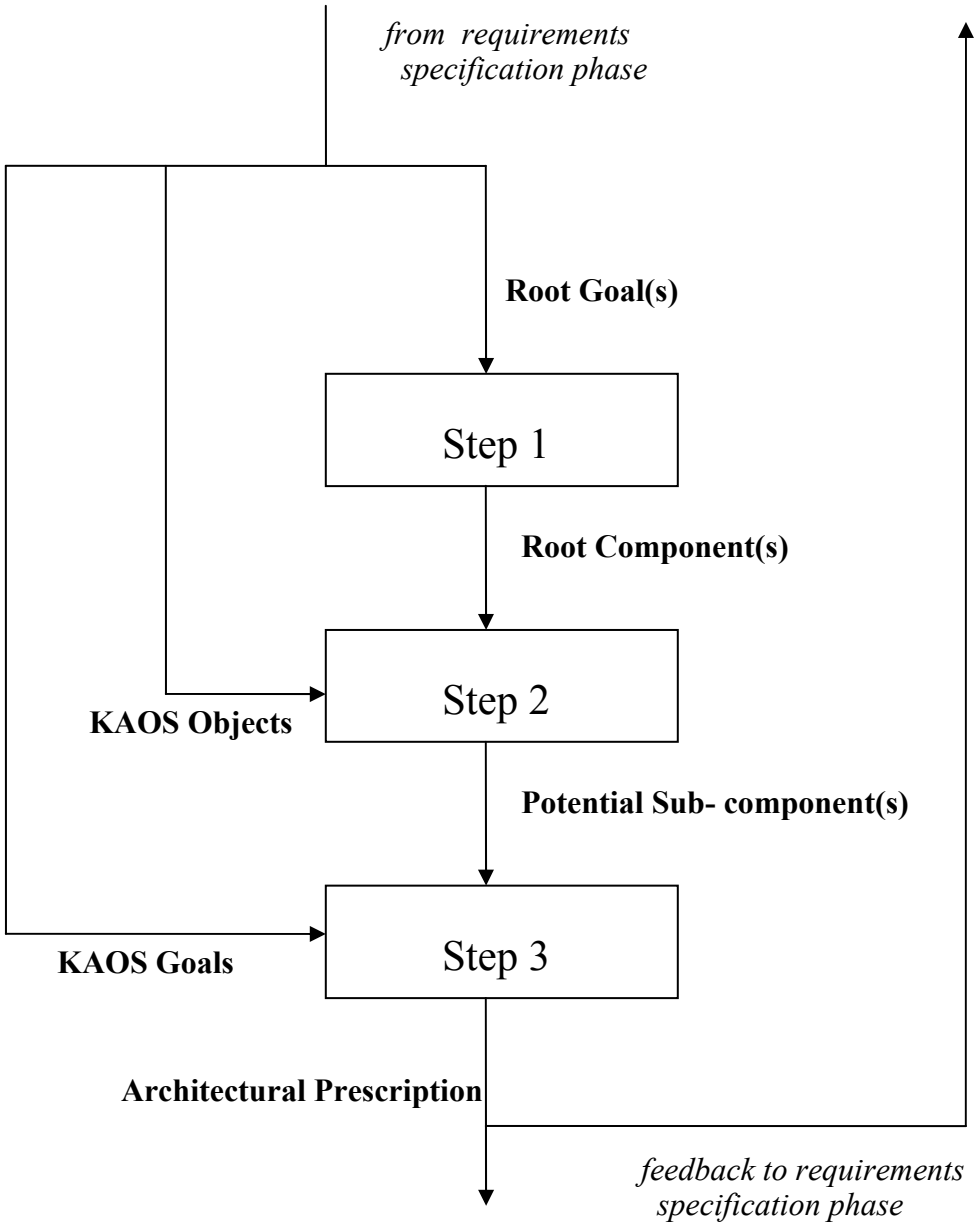

Figure 6: The prescription design process.

The process of deriving the prescription from a domain requirements specification is composed of three steps, and it may be iterated. In the *first* step we derive the basic prescription from the root goal for the system and the knowledge of the other systems it has to interact with. This root goal is either already given, or it can be obtained by induction. In the *second* step we get components that are potential sub-components of the basic architecture from the objects that are in the KAOS specification. In the *third* step we choose the degree of refinement of the goal refinement tree that we consider appropriate, we decide which of the sub-goals are achieved or co-achieved by the software system, and we assign them to the sub-components derived at the previous step. Afterwards, the architectural prescription may be further refined to achieve non-functional properties such as reusability, performance, reliability, etc (i.e. non domain goals).

Now we will illustrate the process with of a practical example. We will consider again the paper selection process for a scientific magazine.

6.1 The First Step of the Methodology

Again, the software system, is co-responsible for the root goal "Maintain[QualityOfTheScientificMagazine]" together with the system composed of the people involved in the process. The software system contributes to achieve the root goal but it cannot achieve it on its own, i.e. it achieves only some of the root goal's sub-goals. The artifact of this step is the very basic architecture for the system that takes into account the fundamental goal and the systems the software system has to interact with. Fig. 7 shows it for our example.

Preskriptor Specification: ScientificPaperManager
Problem Goals Specification: PaperSelectionProcess (KAOS)
Components:

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

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

Figure 7: initial prescription specification

The root component in the tree of Fig. 4, i.e. the software system itself, is refined by the SelectionManager and PeopleConnect components which take into account the root goal and provide the needed interaction with the system made of people. 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. We will fill in the "*Composed of*" field of these two subcomponents after we decide how to refine them at the third step.

6.2 The Second Step

At this step, from the objects that are used in the KAOS specification we derive some potential data, processing and connector components that can implement the *SelectionManager* component we obtained at the previous step. If during next step (the third step) we don't attribute any constraint to these potential components, they won't be part of the system's prescription. That would mean, in fact, that, although they could be used as a particular solution to achieve the goals of the KAOS specification, they won't be necessary to achieve them.

Here is the Preskriptor specification of some candidate objects from the requirements specification in [Cordier 97]:

Component Holds
Type Data
Constraints …
Composed of People*[0,m]*, Document*[1,1]*

Component IsAuthorOf
Type Data
Constraints …
Composed of People*[0,m]*, Document*[1,n]*

Component Supervise
Type Data
Constraints …
Composed of ChiefEditor*[0,m]*, Paper*[0,1]*

Component InChargeOf
Type Data
Constraints …
Composed of AssociatedEditor*[0,m]*, Paper*[1,1]*

Component Evaluates
Type Data
Constraints …

Composed of Evaluator*[0,m]*, Paper*[0,n]*

Component Document
Type Data
Constraints …

Component Paper
Type Data
Constraints …

Component People
Type Data
Constraints …

Component ChiefEditor
Type Data
Constraints …

Component AssociatedEditor
Type Data
Constraints …

Component Author
Type Data
Constraints …

Component Evaluator
Type Data
Constraints …

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

…

Figure 8: Potential components' specification

Since all the components derived from KAOS' specification are data, we need at least a processing component to implement *SelectionManager*. As a particular solution we chose to have just a processing component at this point in the design. We called it *SelectionManagerEngine.* We need also the connectors between this processing component and the previously declared data components. Figure 9. is the specification of these additional subcomponents for *SelectionManager*.

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 …

Component Conn2
Type Connector
Constraints …

…

Figure 9: Potential components' specification – cont.

At next step we will determine which of these subcomponents are really needed to implement *SelectionManager.*

6.3 The Third Step

Now, we will derive the architectural prescription by taking into account goals that are deep in the goal refinement tree. We will show how to put the appropriate constraints on the architectural components of step 2.

At the first refinement of the root goal, the subgoals that the software system contributes to achieve are:

Maintain[OriginalityOfSubmission],

Maintain[QualityOfPublishedArticles],

Maintain[QualityOfPrint],

Achieve[EnoughQuantityOfPublishedArticles].

Let's, for example consider the refinement of the first of these goals. We obtain the following sub-goals:

Maintain[QualityOfEditorialDecisions],

Maintain[PertinenceOfPublishedArticles].

After two further refinements of the first goals of each refinement we obtain (goal G1.1.1.1 in the tree of figure 3):

Avoid[ConflictOfInterestsWithAssociatedEditor]

This goal is the first that can translate into a constraint for the software system only. At this point the software architect has to decide which of the potential components we obtained at step 2 will have to take the responsibility for this goal. There is not only one way to make this choice. In our example, we decided to assign the goal to the *SelectionManagerEngine* and *People* subcomponents. We used the following rationale: in the system to be, *SelectionManagerEngine* will somehow keep track of the different ways the different persons represented by the People data component may know each other. Given our decision, the two constrained components will have to implemented in the next phases of the development process, so that they will indeed achieve this requirement. The existence of this requirement will be a sufficient condition for the existence of the two components. By this we mean that these components have to be in the system 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 on these components, they can be safely discarded.

By proceeding in a similar manner with the rest of the goal refinements, we obtain the first version of a complete APL specification:

Preskriptor Specification: ScientificPaperSelector
Problem Goals Specification: PaperSelectionProcess
Components:

Component SelectionManagerEngine *[1,1]*
Type Processing
Constraints
Avoid[ConflictOfInterestesWithAssociatedEditor],
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 People *[0,n]*
Type Data
Constraints

Avoid[ConflictOfInterestesWithAssociatedEditor],
Avoid[ConflictsWithEvaluator],
Achieve[ListOfPotentialEvaluators]

Component AssociatedEditor *[0,n]*
Type Data
Constraints Avoid[SurchargeAssociatedEditor]

Component Evaluator *[0,n]*
Type Data
Constraints

Avoid[SurchargeEvaluator],

Maintain[CommitedEvaluator]

Component PeopleConnect *[1,n]*
Type Connector
Constraints
Maintain[FeedbackOnPaper],
Maintain[InformationOnEvolutionOfSubmission],
Maintain[ConfidentialityOfPapers],
Maintain[IntegrityOfPapers],
Maintain[ConfidentialityOfSubmission],
Maintain[ConfidentialityOfSensibleDocument],
Maintain[IntegrityOfEvaluation]
Composed of …

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

Component Conn2 *[1,n]*
Type Connector
Constraints
Maintain[IntegrityOfPapers],
Maintain[ConfidentialityOfPapers]

…

Figure 10: Potential components' specification

We want to highlight the fact that the components: ChiefEditor, Author, Knows, Holds, IsAuthorOf, Supervise, InChargeOf and Evaluates, that were potential sub-components at step 2, were taken away from the prescription because they are not necessary to achieve the sub-goals that the system has to achieve. This is only due to the rationale that we took in prescribing the system.

Different architects may use different rationales and produce different prescriptions.

In the general case the effects of non-functional goals on the prescription are additional constraints on the system's components and/or a modification of the system's topology. The latter effect includes the introduction of new components, changing the way components interact and the allowed number of instances for each component. For example, if we have a fault-tolerance goal for some components, in a system that can have at most t faults, the number of instances of the components to achieve fault tolerance will have to be at least $t+1$, and we'll need at least one connector to manage their consistency and manage their interaction with the rest of the system.

Chapter 7: Achieving Non problem domain Requirements

We now introduce a fourth step in the prescription design process, in which an 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 architectural and/or economic reasons.

These goals can be classified as follows: useful architectural properties (such as reusability, reliability, 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 [Perry 00], *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 independency; 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.

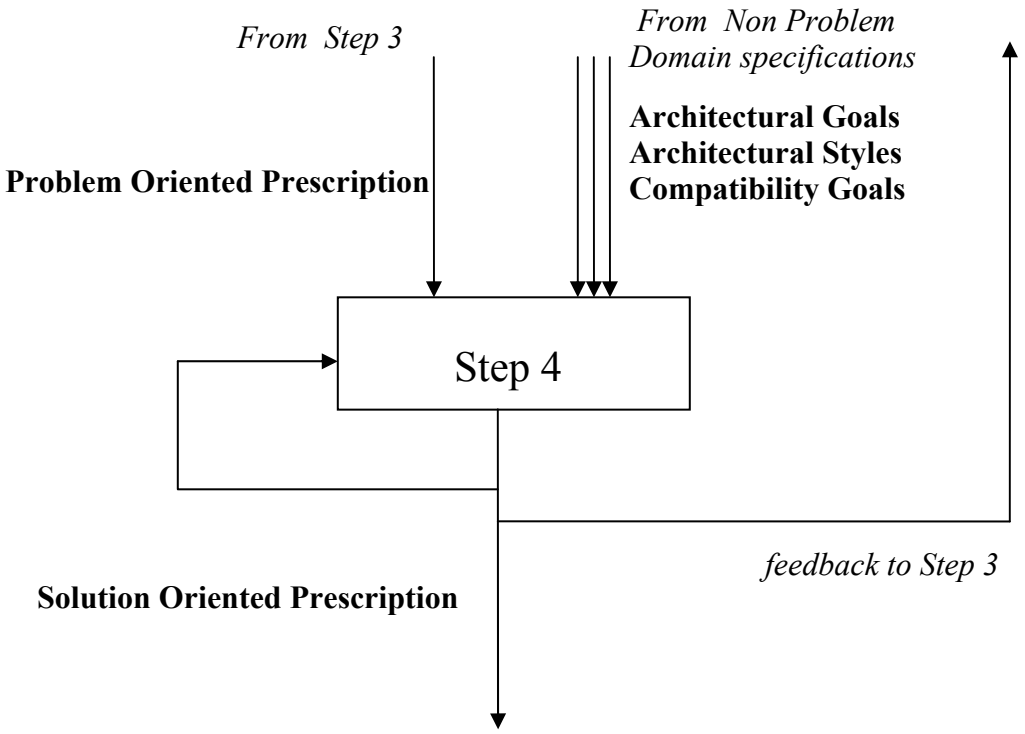

Figure 11: Step 4 in the context of the prescription design

Fig. 11 shows how step 4 interacts with the other steps in the architectural prescription design 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 last 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, that 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 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 we want to implement with different non domain constraints, such as different architectural styles. For this reason we distinguish between the specification of the prescription of after 3, which we call Problem Oriented Architectural Prescription (POAP), from the one after step 4, which we call Solution Oriented Architectural Prescription (SOAP).

Given the Problem Oriented Architectural Prescription for the system and the non-domain driven goals, step four proceeds similarly to step three. It takes as inputs a POAP and the non problem domain goals and gives a SOAP as a result.

In this step the non-domain goals are assigned as constraints to some POAP components and/or the topology of the POAP 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 Architecture Prescription specification is similar to a POAP 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 non problem domain goals (which are generated by the software architects and/or product managers). As we said for a POAP, the Problem Goals specification can be in KAOS or in a similar goals specification language. The specification language used is indicated by parenthesis next to the name of the specification. The Architectural Goals attribute points to the specification of the additional useful properties we want our architecture to achieve. They can be expressed in any goal specification language (it can be KAOS). The Architectural Styles attribute indicates the specification of the styles that the architecture has to satisfy, and the Compatibility Goals attribute takes into account the lowest level goals that the system has to accomplish. Also these last two can be specified with a similar notation.

Figure 12. contains shows of the initial part of the SOAP specification of the example we'll consider in section 7.2:

Preskriptor Specification: MaintainanceSystem
Problem Goals Specification: MaintainanceSystemReqs (KAOS)
Architectural Goal Specifications: /
Architectural Styles Specification: /

Compatibility Goals: CORBA (KAOS)
Components:

…

Figure 12: The beginning of a SOAP specification

In the remainder of this chapter we will illustrate the fourth step with two examples. In paragraph 6.1 we will show how the prescription for the Paper Selection system can be refined to achieve the architectural goal of reusability. In paragraph 6.2 we will consider a new example to show how to take into account a CORBA compatibility goal and how to use requirements specification languages other than KAOS as the starting artifact.

7.1 Reusability Goal Example

How could we modify the prescription obtained in chapter 6. to make it achieve a reusability goal? A way to get a better reusability may be by decomposing a component so that its subcomponents have disjoint sets of constraints. In such a way we get less complex components that can be more easily reused in other systems that have only part of the set of constraints of the original system. We note that the requirements that *SelectionManagerEngine* has to achieve can be split into two main groups: the first six, that provide assistance in choosing assistant editor and evaluators, and the last six, which automate the paper and evaluation documents handling. Since these two groups of services are different in their nature, some other systems we want to develop in the future may

need only one of these kinds of services. We assign the constraints to different sub-components as follows:

Component PeopleSelectionManager *[1,1]*
Type Processing
Constraints
Avoid[ConflictOfInterestesWithAssociatedEditor],
Avoid[SurchargeAssociatedEditor],
Achieve[ListOfPotentialEvaluators],
Avoid[ConflictsWithEvaluator],
Maintain[CommitedEvaluator],
Avoid[SurchargeEvaluator],
Composed of …
Uses PeopleConnect *to interact with* (ChiefEditorAgent, AssociatedEditorAgent, EvaluatorAgent)

Component SelectionProcessManager *[1,1]*
Type Processing
Constraints
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,
…

Figure 13: Component decomposition according to constraints

A way to increase the reusability of connectors is to prescribe different connectors for the interactions of a component with different types of components. In such a way, we reduce the complexity of the connectors, and we

can reuse them in systems in which a component interacts only with a subset of the types of components of the original system. By using this criterion, we can decompose *PeopleConnect* into: AuthorConnect, ChiefEditorConnect, AssistantEditorConnect, and EvaluatorConnect. They provide the user interface respectively with the authors, editors, assistant editors and evaluators.

Component PeopleConnect *[1,n]*
Type Connector
Constraints
Maintain[FeedbackOnPaper],
Maintain[InformationOnEvolutionOfSubmission],
Maintain[ConfidentialityOfPapers],
Maintain[IntegrityOfPapers],
Maintain[ConfidentialityOfSubmission],
Maintain[ConfidentialityOfSensibleDocument],
Maintain[IntegrityOfEvaluation]
ComposedOf
AuthorConnect *[1,n]*,
ChiefEditorConnect *[1,n]*,
AssistantEditorConnect *[1,n]*,
EvaluatorConnect *[1,n]*

Component AuthorConnect
Type Connector
Constraints
Maintain[FeedbackOnPaper],
Maintain[InformationOnEvolutionOfSubmission],
Maintain[ConfidentialityOfPapers],
Maintain[IntegrityOfPapers],
Maintain[ConfidentialityOfSubmission],
Maintain[ConfidentialityOfSensibleDocument],
Maintain[IntegrityOfEvaluation]
…

Figure 14: Connector decomposition according to types of interaction

7.2 CORBA Compatibility Goal Example

How can we use a requirements specification in a non-goal-oriented language as a starting point for our method and how can we specialize a prescription in order to provide CORBA compatibility?

The system we will consider in the following example is part of a Supervision and Control System, namely the one considered in [Coen 00]. The subsystem we consider, the *MaintainanceSystem*, has to detect and manage the failures in the devices of a power plant. The system is specified in the TRIO requirements specification language. TRIO includes a logic language (first order temporal logic) for specifying requirements, and object oriented concepts such as classes and inheritance to specify the elements the requirements refer to. In the following sections we show how we can derive a prescription from TRIO, rather than from KAOS. We also outline the design of a Problem Oriented Architecture Prescription (POAP) and of a Solution Oriented Architectural Prescription (SOAP) that enforces CORBA compatibility, by performing all the steps of the prescription design process we introduced previously.

7.2.1 Step 1

By knowing the fundamental goal of the system, Achieve[DetectFailuresAndMalfunctions], we can design its basic architecture prescription:

Preskriptor Specification: MaintainanceSystem
Domain Goals Specification: MaintainanceSystemReqs (KAOS)
Components:

Component MS *[1,1]*
Type Processing
Constraints Achieve[DetectFailuresAndMalfunctions]
Composed of …
Uses
UserInterface *to interact with* (HumanOperator)
DevicesConnector *to interact with* (Devices)

Component UserInterface *[1,n]*
Type Connector
Constraints Maintain[DetectFailuresAndMalfunctions]
Composed of …
Uses /

Component MeasuringChannel *[1,n]*
Type Connector
Constraints Maintain[DetectFailuresAndMalfunctions]
Composed of …
Uses /

Figure 15: Root POAP specification

7.2.2 Step 2

From the classes in the TRIO requirements specification we derive the following candidate components for the refinement of the basic prescription we obtained at the previous step:

Component IMS *[1,1]*
Type Processing
Constraints …

Component ControlSystem *[1,1]*
Type Processing
Constraints …

Component AlarmManager *[1,1]*
Type Processing
Constraints …

Component GPDB [1,1]
Type Data
Constraints …

Component MeasuringChannel *[1,n]*
Type Connector
Constraints …

Component CS-IMS_Conn *[1,1]*
Type Connector
Constraints …

Component IMS-GPDB_Conn *[1,1]*
Type Connector
Constraints …

Component AM-IMS_Conn *[1,1]*
Type Connector
Constraints …

Component UserInterface *[1,n]*
Type Connector
Constraints Maintain[DetectFailuresAndMalfunctions]
Composed of …
Uses /

Figure 16: Potential subcomponents

7.2.3 Step 3

In order to perform this step we have first to translate the constraints for the Supervision and Control System from TRIO to KAOS. This translation is pretty straightforward, as the specification of a goal in KAOS includes an optional formal definition for the goal. We will get the KAOS goal specification corresponding to a TRIO axiom by assign the TRIO axiom to the formal definition of the KAOS goal.

The following axiom in TRIO states that whenever a self test is started or a command is sent to a device, the IMS has acquired the access rights from the control system:

[axiom1] (test_request(i,MC,test_cmd) $\vee$ command_send(i,dev,dev_cmd)) $\Rightarrow$

access_avail

Here is the corresponding goal in KAOS:

Goal Maintain[AccessConsistency]

FormalDef (test_request(i,MC,test_cmd) $\vee$ command_send(i,dev,dev_cmd)) $\Rightarrow$

access_avail

A similar kind of translation to KAOS could be applied to other formal requirements specification. This means, we can use our architectural prescription design process with virtually any goal oriented or formal requirements specification languages.

Despite the fact that a TRIO specification does not provide different degrees of requirements refinement, we can achieve them in the corresponding KAOS specification by generalizing the goals we derived from TRIO. This means we can require, at step three, a lower degree of constraint for the system than the one directly induced by the TRIO specification. For simplicity, in our example we'll consider the goals derived directly from TRIO.

Here is part of the Problem Oriented Architecture Prescription (POAP) for the system:

Preskriptor Specification: MaintainanceSystem
Problem Goals Specification: MaintainanceSystemReqs (KAOS)
Components:

Component IMS *[1,1]*
Type Processing
Constraints Maintain [AccessConsistency], …

Component ControlSystem *[1,1]*
Type Processing
Constraints Maintain [AccessConsistency], …

Component GPDB *[1,1]*
Type Data
Constraints …

Component MeasuringChannel *[1,n]*
Type Connector
Constraints …

Component CS-IMS_Conn *[1,1]*
Type Connector
Constraints Maintain [AccessConsistency], …

Figure 17: A refined POAP specification

7.2.4: Step 4

Now we want to design a Solution Oriented Architecture Prescription (SOAP) that guarantees that the system will be CORBA compatible. A SOAP is a kind of architecture specification that is at a higher level of abstraction than the specification used in [Coen 00] and [Prad 00]. In fact, they consider a design that is at the CORBA level of abstraction, while a SOAP is still at the architecture prescription level. This means a SOAP leaves up to the successive design artifact(s) (architecture description, low level design) the decisions on how to implement the components in order to satisfy their constraints. These decisions include choosing the already existing applications or components, classes, attributes, operations and protocols to implement each of the prescription's components.

We assign the CORBA_compatibility only to those components that will have to interact with other CORBA components. By CORBA_compatibility we

mean that the implementation of the component will be according to CORBA, i.e. it will be implemented as one (or more) CORBA object(s). A component that contributes to achieve different domain goals, only some of which can take advantage of CORBA, can be split into two sub-components, so that CORBA will be prescribed for a component only when it's strictly required.

Here is the outline of a CORBA compatible SOAP for the *MaintenanceSystem*:

Preskriptor Specification: MaintainanceSystem
Problem Goals Specification: MaintainanceSystemReqs (KAOS)
Compatibility Goal Specification: CG Name (Specification Language)
Components:

Component IMS *[1,1]*
Type Processing
Constraints Maintain [AccessConsistency], CORBA_compatibility, …

Component ControlSystem *[1,1]*
Type Processing
Constraints Maintain [AccessConsistency], CORBA_compatibility, …

Component GPDB *[1,1]*
Type Data
Constraints CORBA_compatible, …

Component MeasuringChannel *[1,n]*
Type Data
Constraints …

Component CS-IMS_Conn *[1,1]*
Type Connector
Constraints Maintain [AccessConsistency],

…

Figure 18: A SOAP specification for CORBA compatibility

Chapter 8: Conclusion

In this thesis we have advocated the use of structured, systematic and rigorous techniques to perform the different phases of the software development process. We focused on what we consider the most important of these phases: the one that bridges requirements and architecture. To achieve the method we are proposing we have introduced the concept of an architecture prescription language (APL), which specifies a high level architecture (prescription). An architecture prescription provides the basic framework of the system to achieve the requirements. It specifies the structure of the software system and its components with the terms of the application domain. We then described the prescription design process and illustrated it with practical examples.

There is a growing community of researchers studying the interplay between the requirements and the architecture. A. Egyed et al., in [Egyed 01], developed a technique to pass from requirements specified in the WinWin language to an architectural model for the system called CBSP. Their technique, while providing a framework to pass from requirements to architecture, still provides little guidance to the architects. Moreover it bases the choices of the components and their importance on a vote by all the stakeholders involved. We don't believe that a democratic vote is the best way to design an architecture. Other researchers have produced object-oriented techniques to pass from

requirements to architecture. These techniques, though, provide only a little guidance to the architect and they are limited to an object oriented architectural design.

Our goal is to design a systematic and rigorous technique to pass from the requirements to an architectural prescription. The rigor is necessary to make sure that none of the requirements are neglected, and that we don't introduce any useless one

As for our future work, we will investigate how particular non-functional goals affect architectural prescriptions. We'll look for ways to generate a prescription bottom-up (from more constrained to less constrained specifications). Most importantly, we will perform experiments to validate our hypotheses and gain new insights. Finally, we plan to build a tool set based on our methodology.

Appendix

Preskriptor Specification: [Prescription's name]
(Domain Goals Specification: [Domain specification's name] ([Language's name]))$^{?}$
(Architectural Goals Specification: [Architecture specification's name] ([Language's name]))$^{?}$
(Architectural Styles Specification: [Styles specification's name] ([Language's name]))$^{?}$
(Compatibility Goals Specification: [Compatibility specification's name] ([Language'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 that place. "*" means that the immediately preceding expression can be present from zero to an arbitrary number of times. "+" is the same than "*" except that it has to be present at least once. "?" means the expression can be present either zero or one time. The new symbol "&" means that the expression is required only for the specification of the components that belong to the first layer of the components refinement tree.

References

[Boehm 88] Boehm, B.W., "A Spiral Model of Software Development and Enhancement". IEEE Computer, 21, 5, May 1988, pp. 61-72

[Coen 00] "Using TRIO for designing a CORBA based application", Concurrency: Practice and Experience, August 2000

[Cordier 97] Cordier, C., and Van Lamweerde, A., "Analyse des constraintes de sécurité puor la gestion électronique d'une revue scientifique", Universite Catholique de Louvain, 1997

[Egyed 01] Egyed, A., Grünbacher, P., and Medvidovic, N. "Refinement and Evolution Issues in Bridging Requirements and Architectures," Proceedings of the 1st International Workshops From Requirements to Architecture (STRAW), co-located with ICSE 2001, Toronto, Canada, May 2001, pp. 42-47

[Jack 95] M. Jackson. The world and the machine. In Proceedings of the 17th International Conference on Software Engineering, Seattle, Washington (USA), April 1995. Keynote speak

[Lam 95] Van Lamweerde, A., Darimont, R., and Massonet, P., "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

[Lam 98] 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

[Perry 92] Dewayne E. Perry, Alexander L. Wolf, "Foundations for the Study of Software Architecture", Software Engineering Notes, ACM SIGSOFT, October 1992, pp. 40-52

[Perry 00] 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

[Prad 00] A.Coen Porisini, M. Pradella, M. Rossi, D.Mandrioli “A Formal Approach for Designing CORBA based Applications” Proc. ICSE, Limerick, pp. 188-197, June 2000