Architectural Prescriptions for Dependable Systems

Manuel Brandozzi
UT - ARISE
Advanced Research In Software Engineering
The University of Texas at Austin
mbrandozzi@ece.utexas.edu

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

ABSTRACT

In this paper, we advocate the enforcement of dependability requirements at the architectural design level of a software system. We illustrate how to achieve this by using our methodology, which provides a guideline of how to design an architectural prescription from a goal oriented requirements specification of a system. We distinguish between separation, additive and integral non-functional requirements, and discuss their different effects on a prescription. In particular, additive non-functional requirements provide separation of concerns by only adding to the system some new components to achieve them. Dependability requirements are a particular kind of non-functional requirements and often they are additive.

1. INTRODUCTION

Experience has shown that, for complex software systems, it's very important to take into account the non-functional requirements as early as possible during the systems' design process. This means that they should be taken into account already at the architectural design level. By doing so, it's possible to understand what the implications of this kind of requirements are on the high level components and on the high level structure of the systems.

Dependability requirements are a particular type of non-functional requirements. In this paper we adopt a broad definition of dependability and we intend it to "embrace all those aspects of behavior upon which the user of a system might need to place dependence: it thus includes reliability, safety, availability and security" [3].

Another way that an architectural prescription favors the design of dependable systems is by enabling the reuse of the high level design of systems that, having been already deployed, have been demonstrated to be dependable. A prescription allows the architect to reuse all the components and the topology that derive from particular goals (i.e. requirements), including dependability requirements. Generally, a brand new system design has a higher likelihood of failure than a well tested one.

Before illustrating our approach for dependability enforcement at the architectural level in section 4, we provide an introduction to goal oriented requirements and to architectural prescriptions in sections 2 and 3 respectively. In section 5 we illustrate our approach with an example and we summarize our contributions and discuss future work in section 6.

2. GOAL ORIENTED REQUIREMENTS SPECIFICATIONS AND KAOS

Goal oriented requirements specifications 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. KAOS is the goal oriented specification language, introduced by A. van Lamsweerde [2], that we used in our methodology.

The 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 objects. Operations 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 of the tree 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. Here is an example of goal declaration in KAOS:

Goal Maintain[AuthorizedAccessesOnly]
InstanceOf SecurityGoal
Concerns StockValues, BankerActor
ReducedTo
ConfidentialityOfAccessPassword,
ConfidentialityOfTransmittedStockValues
InformalDef
Access passwords must remain confidential. Stock values information has to be released only to those providing the correct passwords.

Example 1: 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 refined. Finally, there is *InformalDef*: the informal definition of the goal. There can also be an optional attribute *FormalDef*, which contains a formal definition of the goal (that can be expressed in any formal notation such as linear temporal logic).

3. ARCHITECTURAL PRESCRIPTIONS AND PRESCRIPTOR

An architectural prescription [1] lays out the space for the system structure by selecting the architectural components (processes, data, and connectors), their relationships (interactions) and their constraints. In a prescription, the fundamental characterization of components is given by the goals they are responsible for (that are their constraints). Components are further characterized by their type: processing, data or connector. The processing components are those that 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 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.

Example 2 contains the architecture prescription of a data component specified in Preskriptor, our architectural prescription language that takes KAOS requirements specifications as starting point.

Component StockValues *[1, 1]*
Type Data
Constraints Maintain[LatestStockValuesInfo], …
Composed of DB *[1,1]*, Server *[1,1]*
Uses MarketConnect *to interact with*
StockMarket

Example 2: a component's specification in Preskriptor.

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. Note that the semantics of any component in Preskriptor is provided by its constraints and only by them. 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 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 a particular connector. The symbol "/" means no attribute and, since now, we will omit the fields whose value is none.

Without going into the details of how to get a prescription from the requirements [1], it's important to know that at the beginning some candidate components for the architecture are proposed, then the functional goals first, and non-functional goals afterwards, are assigned, one at a time, to a subset of the potential components. Those components who do not contribute to the achievement of any goal are discarded from the system. The next section explains in some detail how to account for non-functional requirements in an architectural prescription.

4. NON-FUNCTIONAL REQUIREMENTS IN PRESCRIPTION DESIGN

Taking into account Non-Functional Requirements (NFRs) while designing an architectural prescription has, in the most general case, three kinds of effects on the already designed prescription of a system:

1) The introduction of new components.
2) The transformation of the system's topology, i.e. a change on the relationships among the system's components.
3) The further constraining of already existing components.

Some non-functional requirements allow for separations of concerns among the architecture components; other requirements, instead, are spread throughout the code: they reach every component of the system like blood vessels reach every cell of our body.

We denote those NFR that enable separation of concerns with respect to an architecture as *Separation Non-Functional Requirements* (SNFR). SNFR are those requirements that can be achieved by further constraining, adding new components and/or by transforming the topology of only a precisely identifiable subset in strict sense of the architecture's components. By "precisely identifiable" subsystem we mean that the subsystem can be characterized by a property. By subsystem "in strict sense" we exclude the complete system, case in which we don't achieve separation of concerns. A precisely identifiable subsystem in strict sense can be, for example, a single component of the system. This happens in the case of a performance goal when a single component is the bottleneck for computation. Another example of a precisely identifiable subsystem is the set including all the connectors from a particular component, and the component itself, like in the fault tolerance example that we will illustrate in next section.

The simplest SNFRs are those that in some architectures can be achieved by only adding to the system new components and the relationships of these new components with other components, i.e. by composing some existing components with new ones without changing the constraints of any of the old components. We denote this kind of NFRs as *Additive Non-Functional Requirements* (ANFRs).

Those NFRs that are not SNFRs are denoted as Integral Non-Functional Requirements (INFRs). These requirements affect the entire system or a subset of the system for which no characterizing property can be found, i.e. the system is not precisely identifiable. A way to achieve this other kind of requirements is by making all components conform to a particular style. An example of INFR is the goal for a system is to be composed by only components that conform to CORBA. No matter what, this requirement has to be added as a constraint to all the system's components.

In general, whether an NFR is integral, separation, or additive depends on the architecture on which we want to achieve it. It also may depend on the level of refinement of the architecture. In fact, what at a finer resolution of an architecture is a clearly identifiable subset in strict sense may become the whole set of the system's components at a coarser refinement.

5. ENFORCING DEPENDABILITY AT THE ARCHITECTURAL LEVEL

Let's see, with the aid of an example, how a dependability requirement (in this case fault tolerance) can be handled by an architectural prescription.

Any computer network may have, even in absence of catastrophic events, a certain number of machines that crash or become inaccessible. Let's consider the case of a distributed system, that runs on such a network, and that contains a data component whose accessibility at any time is vital. The data component can, for example, contain the value of the stocks managed by an investment bank. It's vital that the bankers can access at any time the current value of a stock. Not being possible to do so could cost to the bank thousands of dollars, if not millions!

This kind of fault tolerance problem has been widely studied in the distributed systems community and a standard solution to it is the following. Suppose that in a network with x nodes there can be at most t (with t < x) nodes that can fail at the same time. We can achieve a fault-tolerant real-time access to the vital data object StockValues, by having, at least, t+1 copies of the object stored in t+1 different nodes. To guarantee this we also need some protocol that manages the access to the object from outside the network, and that updates of the copies of the object to achieve consistency among them.

Example 3. contains the prescription of a simple distributed system. This is the prescription of the system before we take into account the fault tolerance goal.

Component StockValues *[1, 1]*
Type Data
Constraints Maintain[LatestStockValuesInfo], …
Composed of DB[1,1], Server[1,1]
Uses MarketConnect *to interact with* StockMarket

Component BankerClient *[0, n]*
Type Processing
Constraints ...
Uses
StockValuesAccess *to interact with* V
BankerUserInterface *to interact with*
BankerActor

Component StockValuesAccess *[0, n]*
Type Connecting
Constraints Maintain[AuthorizedAccessesOnly]

Example 3: prescription for a stock values information system

In the prescription of example 3 we have only one copy of the data component *StockValues*, the component storing the latest values of the stocks belonging to different markets that is updated by using *MarketConnect* connecting it to the stock markets. The prescription allows any number n of components *BankerClient* to be instantiated. *BankerClient* is the piece of software running on the bankers' machines.

The prescription requires the system to have communication between *BankerClient* and *StockValues* through connector *StockValuesAccess*, that has to achieve the security goal *Maintain[AuhorizesAccessesOnly]* (defined in Example 1.) together with other goals (such as mutual exclusion) not included here for simplicity. Given a particular choice of implementation of connector StockValuesAccess, the low level design may instantiate the connector only once for all the n Clients, instantiate it n times, or any other number of times between 0 and n.

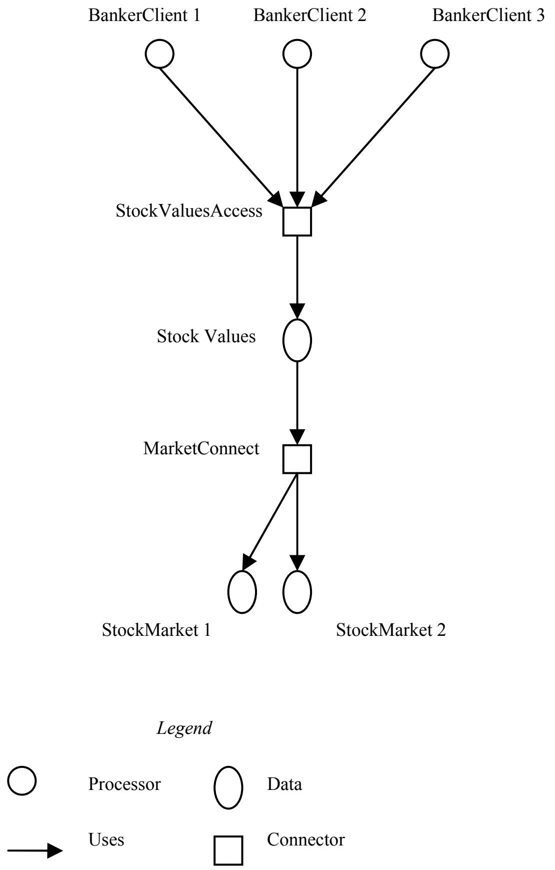

Figure 1. Topology graph of the prescription in example 3.

Figure 1. contains the graphical representation of the topology of an instantiation of the same prescription. It's the topology of an instantiation of the prescription because for each component a given number of instances has been chosen. For example, there are only three clients, rather than an indefinite number; and there is only one instance of connector *StockValuesAccess* for the interaction of all the clients with *StockValues,* rather than an indefinite number of connectors that is at most equal to the number of *BankerClient* components. Graphical representations are useful to better understand the topology that is specified by the prescription.

In any prescription graph, the arrow representing the *Uses* attribute goes from the component C, that needs some information from the interaction, to a connector CN that makes the interaction possible; and then from the connector CN to those components that provide the information needed by C.

Example 4. shows the same prescription after it has been transformed to account for the non-functional goal of fault tolerance of the component StockValues.

Component StockValues *[t+1, n]*
Type Data
Constraints Maintain[LatestStockValuesInfo], …
Composed of DB[1,1], Server[1,1]
Uses
MarketConnect *to interact with* StockMarket
InterCopyCoordinator *to interact with*
StockValues

Component InterCopyCoordinator *[1, n]*
Type Connecting
Constraints Maintain[FaultTolerance]

Component StockValuesAccess *[0, n]*
Type Connecting
Constraints Maintain[AuthorizedAccessesOnly]

Component StockValueFTAccess *[1,n]*
Type Connecting
Constraints
Maintain[FaultTolerance],
Maintain[AuthorizedAccessesOnly]
Composed of
IntercopyCoordinator *[1,n]*, StockValuesAccess *[0,n]*

Component BankerClient *[0, n]*
Type Processing
Constraints …
Uses
StockValuesFTAccess *to interact with*
StockValues
BankerUserInterface *to interact with*
BankerActor

Example 4: prescription for a stock values information system with fault tolerance

Like many dependability requirements, the non-functional fault tolerance requirement is an ANFR. It's an ANFR because it can be assigned as a constraint only to the new connector *InterCopyCoordinator,* which coordinates the now multiple copies of component *StockValues*. This is an example of achieving an ANFR via connectors; another such example can be found in a system developed by the DSSA group [5], case in which the NFR is performance.

The system specified by the new version of the prescription has to have at least t+1 (t being the maximum number of faults) copies of component *StockValues*, rather than only one like in its pre-fault tolerance prescription. *StockValues* is now using the newly added connector *InterCopyCoordinator*. To achieve the fault tolerance goal, among the other things, this connector will have to make it sure that, at any time, there are at least t+1 copies of *StockValues*. Also, it has to assure that the different copies are, somehow, kept consistent at least from the perspective of the rest of the software system. The *BankerClients* interacting with component *StockValues* must always get the latest value for the stocks. The access to *StockValues* by two or more clients at the same time has to abide to the same mutual exclusion policies that held when only an instance of *StockValues* was in the system. We designed the prescription so that *InterCopyCoordinator* keeps the topological transformations transparent to *BankerClient*. The only change in *BankerClient*'s specification is that now this component uses connector *StockFTValuesAccess* (resulting from the composition of *InterCopyCoordinator* and *StockValuesAccess*) to interact with *StockValues*, rather than using *StockValuesAccess*. It's the *InterCopyCoordinator*'s subcomponent of *StockFTValuesAccess* that hooks into *StockValues* to guarantee that *BankerClient* always gets the updated values of the stocks.

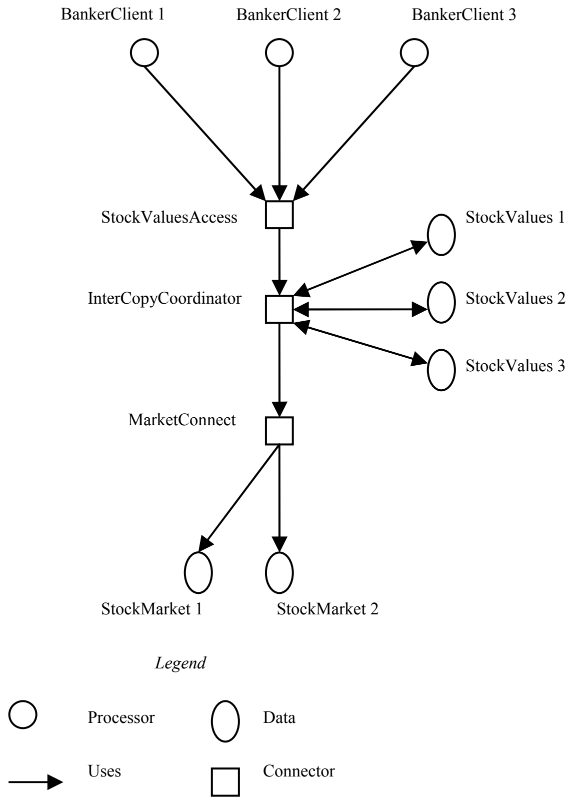

Figure 2. Topology graph of the prescription in example 4.

In a particular implementation of the prescription in a latter phase of the development process, *InterCopyCoordinator* may take care of the creation of t+1 copies at start-up, as well as creating new copies, moving the existing ones, or remove copies whenever some node fails or to save on communication costs like in illustrated in [4].

The effects of the topological transformation are evident if we have a look the topology graph of the new prescription in figure 2. Here, the graph is the same than the one of figure 1. apart from having substituted the single component *StockValues* with a more complex component that is composed by the different *StockValues* instances (three in the example) and the *IntercopyCoordinator* used by them. The double edged arrows are a syntactic shortcut to make the graph more elegant: they represent all the arrows that depart from and go to a particular component.

6. CONCLUSION

Dependability requirements are a large subset of non-functional requirements. To better achieve them and manage their changes they should be taken into account already at the architectural design level. We provided an overview of our methodology to design an architectural prescription given a set of goal oriented requirements specifications.

The requirements can either be functional or non-functional. Separation Non-Functional Requirements (SNFRs) enable separation of concerns in achieving them. Their effects are limited to a subset of the system identifiable by a property (like the set of connectors outgoing from a particular component). In particular, we illustrated with an example how a fault tolerance requirement for an object in a network (that is an ANFR, an easier case of SNFR) can be achieved by a given architecture. This was done by introducing in the architecture a new connector and modifying the topology of the system locally to one of its components. Many other dependability requirements, including security, performance and other kinds of fault tolerance can be ANFRs with respect to many architectures.

Our future work will be aimed at finding out domain independent ways to compositionally transform the prescription of a system to account for ANFRs and at developing a tool to do so automatically.

5. REFERENCES

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

[2] 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, 194-203

[3] Littlewood, B. Evaluation of software dependability. Computing Tomorrow: Future Research Directions in Computer Science, Book, I. Wand and R. Milner, 1996

[4] Johnson, G., Singh, A. Stable and fault-tolerant object allocation. Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon 259-268

[5] Tracz, W. Domain-Specific Software Architecture Pedagogical Example. ACM Software Engineering Notes, July 1995, 49-62.