Deriving Architecture Specifications from KAOS Specifications: a Research Case Study [⋆]

Divya Jani[1], Damien Vanderveken[2], and Dewayne Perry[1]

[1] Empirical Software Engineering Lab, ECE, University of Texas at Austin
divyaj@mail.utexas.edu
perry@ece.utexas.edu
[2] Dept. d'ingenierie informatique,Universite catholique de Louvain
damien.vanderveken@swift.com

1 Introduction

The most difficult step in the design process of a system is clearly the transition from the requirements to the architecture. Requirements obtained from the various stakeholders must be transformed into an architecture that can be understood by developers. The power plant system we use in this study was derived from [1, 2]. We first created a goal-oriented requirements specification from the information available using the KAOS requirement specification language [3–5]. Since the description was not complete we often had to make do with inadequate data.

The first method used was developed by Axel van Lamsweerde (University of Louvain - Belgium) and is described in [6]. The various steps are explained in detail in Section 3.1 We describe some of the problems encountered during the derivation process. The second method used was that of Dewayne Perry and Manuel Brandozzi (University of Texas at Austin) [7–9]. The resulting architecture and some of the derivation issues are described in Section 3.2.

After obtaining both architectures we compared them and suggested some further work. In the case of the Perry/Brandozzi method we have made improvements to solve the problems we encountered and added the consideration of styles and patterns for non functional properties.

This case study [10] was structured as follows. First the authors together created the KAOS specification of the problem. Second Jani and Vanderveken then used the two methods to transform this specification into architecture specifications with Perry acting as mentor, arbitor and oracle in recording process issues and providing direction at critical points in the process. The authors together evaluated and compared the results.

[⋆] The research was supported in part by NSF CISE Grant CCR–0306613 "Transforming requirement specifications into architectural prescriptions"

2 Requirements derivation using KAOS

2.1 Goal Model

Given the fact that KAOS is a goal-oriented requirement specification method we logically began by trying to extract the goals of the system. A definition of the system was implicitly given in [1]. However the description of the powerplant monitoring system provided was partial and lacked details. So, throughout the requirement extraction process, we had to rely on experience and our common sense to create requirements that are as realistic as possible.

The following steps were followed to build the goal model. First of all, the informal definition of goals mentioned in [1] were carefully written down. From that, a goal refinement tree was built and completed by a refinement/abstraction process. The version we obtained at that point was still totally informal. Temporal first-order logic [11] was then used to formalize the goals and to ensure our refinement tree was correct, complete and coherent. The use of refinement patterns as described in [3] served as guidance. The milestone-driven pattern in particular was applied numerous times. It prescribes that some milestone states are mandatory in order to reach a final one. This pattern is presented in fig 1. The patterns were a great help to track and to correct incompleteness and incoherence. Furthermore they enabled us to save a huge amount of time by freeing us to do the tedious proof work.

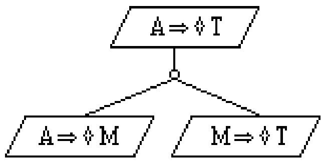

Fig. 1. Milestone refinement pattern

Because of the iterative nature of the requirements gathering process, the goal model underwent subsequent changes. The reasons for that varied: coherence between the different models forming the KAOS specifications, enhancements, simplifications,etc.

The goal refinement tree is globally structured in two parts. This shape reflects the two main goals the system has to ensure to monitor the powerplant. The occuring faults have to be detected and the alarms resulting from those faults have to be managed. The roots of the two resulting subtrees are respectively *FaultDetected* and *AlarmCorrectlyManaged*. They are subsequently refined using the various patterns until the leaf goals are assignable to a single agent from the environment or part of the software.

As an illustration of the use of the milestone refinement pattern let's consider the goal *AlarmRaisedIfFaultDetected* with its formal definition

$$\big(\forall f : Fault, \exists! l : \mathit{Location}, \exists! a : \mathit{Alarm}\big)\big(\mathit{Detected}(f, l) \Rightarrow \Diamond \mathit{Raise}(f, a)\big) \quad (1)$$

This goal is refined using the milestone refinement pattern by instantiating the parameters as follows:

$$A : \big(\forall f : Fault, \exists! l : \mathit{Location}\big)\big(\mathit{Detected}(f, l)\big) \quad (2)$$

$$M : \big(\exists fi : FaultInformation\big)\big(f \equiv fi \wedge Transmitted(fi, PRECON, ALARM)\big) \quad (3)$$

$$T : \big(\forall fi : FaultInformation, \exists! a : \mathit{Alarm}\big)\big(\mathit{Raised}(fi, a)\big) \quad (4)$$

The application of that pattern in particular results here from the fact that the information concerning the detected faults has to be transmitted to the ALARM to enable it to raise the proper alarm. This intermediate state is a necessary step to reach the final state, i.e., raising the alarm.

To have a system as robust as possible various goals were added to the goal diagram. Among these added goals, one class takes care of the correct working of all the sensors and ensures the data provided is consistent and coherent. The goals *SanityCheckPerformed* and *ConsistencyCheckPerformed* belong to this class. Another class – represented by the goal *DataCorrectlyUpdated* – makes sure the updates are well performed by the database. The purpose of some goals is to maintain the powerplant in a consistent state (e.g., *FaultStatusUpdated*, *AlarmStatusUpdated*). Communication has also been constrained in order to prevent any transmission problems and results in the refinement of the goal *DataTransmittedToDB* where refinement is shown in Fig. 2.

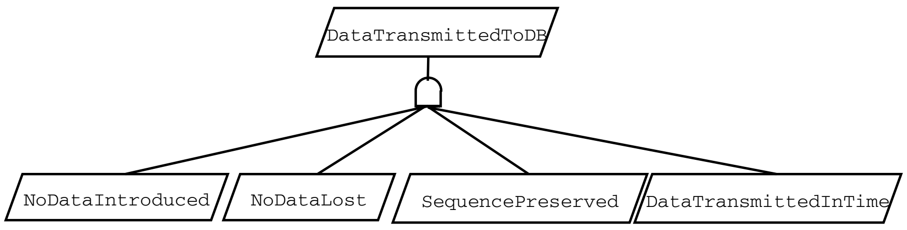

Fig. 2. Communication refinement subtree

The three first subgoals ensure the correctness of the transmission while the last one sets a time limit. This constraint varies througout the system depending on the importance of the communication channel. The *FaultInformation* has to be transmitted from PRECON to ALARM within 1 second while answer a request can take a little longer – 5 seconds. The three first subgoals have been formally refined as followed [3]:

$$\begin{aligned} &NoDataIntroduced: \\ &\big(\forall x: \mathit{Data}\big)\big(Transmitted(X,\_,\_) \wedge x \in Transmitted(\_) \Rightarrow x \in X\big) \quad (5) \\ &NoDataLost: \\ &\big(\forall x: \mathit{Data}\big)\big(x \in X \wedge Transmitted(X,\_,\_) \Rightarrow x \in Transmitted(\_)\big) \quad (6) \\ &SequencePreserved: \\ &\big(\forall x, y: \mathit{Data}, \exists u, v: \mathit{Data}\big)\big(x, y \in X \wedge Transmitted(X,\_,\_) \\ &\quad \wedge Before(x,y,X) \Rightarrow u, v \in Transmitted(X) \wedge \\ &\quad Before(u,v,Transmitted(X)) \wedge x = u \wedge y = v\big) \quad (7) \end{aligned}$$

They prescribe that no alteration has occured on the data transmitted i.e., no data has been introduced or lost and the sequential order has been preserved.

The formal definition of the last subgoal depends on the time constraint. If we consider for example the transmission of a *FaultInformation* – which has the strongest time constraint – the formalization is:

$$\begin{aligned} &DataTransmittedWithinTimeConstraint: \\ &\neg Transmitted(fi, PRECON, ALARM) \quad \Rightarrow \Diamond_{\leq 1s} \\ &\quad Transmitted(fi, PRECON, ALARM) \quad (8) \end{aligned}$$

-

2.2 Object Model

Entities present in the objects were first derived from the informal definition of the goals. All the concepts of importance were modelled either under the form of an object or of a relationship. Attributes were then added to the different entities to characterize them. Some of the attributes were extracted from the problem definition but most of them proceed necessarily from the underlying domain from two main reasons.

First, certain goal definitions need the presence of specific attributes. For example the attribute *WorkCorrectly* of *Sensor* was needed by the goal *Sanity-CheckPerformed*.

Second, the definition of the properties of the various entities – expressed by invariants – requires specific attributes. As an illustration consider the following invariant of the object *Alarm* which expresses that all the alarms still active cannot have a deactivation time:

$$\mathit{Activated} = \mathit{true} \Rightarrow \mathit{DeactivationTime} = \mathit{null} \quad (9)$$

The purpose of certain attributes is to prepare for change. The reconfiguration function was not taken into account in the elaboration of the different

[3] X stands for *SensorInformation*, *FaultInformation*, *AlarmInformation*, *FaultDiagnosis* and *AlarmDiagnosis*

models due to lack of time. However we believe that basically the only effect will be to modify the allowed range of temperature and pressure. Attributes representing the minimum, the maximum and desired value of both pressure and temperature were consequently added to the objects *SteamCondenser* and *CoolingCircuit*.

Last, a few attributes were added to build a more complete model. The justification was common sense. Among these are the attributes *Type* and *Power* of the object *PowerPlant*.

The last step in the elaboration of the goal model was the formalization of the domain invariants characterizing the differents entities. The model was refined many times due to the iterative nature of the requirement extraction process.

The main characteristic of the model is that two different levels of representations are used for the concepts *Sensor*, *Fault* and *Alarm*. The first level refers to the object itself while the second one refers to its representation in the software. This distinction was introduced for robustness reasons. In fact it enables us to manage the case where the representation of the object is not correct which would be unfortunate but can happen. The two levels are constrained by an invariant prescribing that all the attributes have to be identical.

The representation of the three main objects – Sensor, Fault and Alarm – are linked together by a diagnosis relationship. The information provided by the sensor permits the detection of the faults and the description of a fault is the rationale for the raising of an alarm. Consequently the relationship *FaultDiagnosis* links *SensorInformation* and *FaultInformation* while *AlarmDiagnosis* links *FaultInformation* and *AlarmInformation*. Those two relationships are one-one. It is a modelling choice. We chose that a fault is the result of one and only one error detected by one sensor and that each fault raises one and only one alarm. The resulting simplicity and the ease of traceability is the reason for that.

2.3 Agent Model

The definition of the agents was extracted mostly from [1, 2]. We drew inspiration from the existing agents as well. Each leaf goal from the Goal Model was assigned to an agent. We made sure that every agent had the capacity to assume the responsibility for that goal. By capacity we mean that every agent could monitor or control, every single variable appearing in the formal definition of a goal the agent has to ensure. For further details refer to [5].

However a new agent was introduced : `MANAGEMENT UNIT`. Its purpose is to ensure that all the sensors are working properly. It was added for robustness.

Finally the operations needed to operationalize the differents goals were assigned to their responsible agents. This step will be explained later in the Operation Model section.

The agents `PRECON`, `ALARM`, `COMM`, `DB` and `Sensor` come from [1] though their names are different from there. `PRECON` is in charge of the detection of all the faults that might occur either in the cooling circuit or in the steam condenser. `ALARM` takes care of the alarm management. `COMM` ensures the reliability and the performance of all the communcication throughout the system. `DB` stores all the data persistently and answers all the requests concerning current values of the sensors, faults and alarms. The `Sensor` agent acquires the data from the field. The additional agent – `MANAGEMENT UNIT` – checks the sensors to see if they work properly.

The agents belong to one of two different categories: they are part of the software-to-be or part of the environment. For example, `PRECON` belongs to the former while `Sensor` belongs to the latter. This distinction in agents results also in goal differentiation. In fact the goals assigned to environment agents are expectations while the others are requirements. This led us to the introduction of the `MANAGEMENT UNIT` agent. `Sensor` is an environment agent and so all the goals assigned to it are expectations. But obviously we canot assume that the goals `SanityCheckPerformed` and `ConsistencyCheckPerformed` will be true without the intervention of reliable software devices. Moreover these kinds of tests should not be the responsibitlity of the `Sensor` from a conceptual point of view.

2.4 Operation Model

The operation model was the the last one to be constructed because it relies on a precise formal definition of the goals. The operations contained in the model were derived in such a way that they operationalize some goal present in the goal model. A complete operationalization of a goal is a set of operations (described by their pre-, trigger- and postconditions) that guarantee the satisfaction of that goal if the operations are applied. That is where all the difficulty lies: finding complete operationalizations. We extensively used the operationalization patterns described in [4] to derive complete operation specifications. It enabled us to save a lot of time on proofs. We found the application of the operationalization pattern very systematic.

Two patterns were particularly useful and we used them numerous times. The first one is the bounded achieve pattern described in Fig. 3. Its applicabilty condition (i.e., $C \Rightarrow \Diamond_{\leq d} T$) is pervasive. In fact most of our system's goals have that form. The operation specification prescribes that $\neg T$ becomes T as soon as $C \wedge \neg T$ holds for $d-1$ time units. It is then straightforward to see that such a specification operationalizes the goal $C \Rightarrow \Diamond_{\leq d} T$.

The second most useful pattern was the immediate achieve pattern described in Fig. 4. Its applicability condition prescribes here that the final state T has to be reached as soon as C becomes true. In this case it is a bit more difficult to see why the satisfaction of the two operations guarantee the satisfaction of the goal (the interested reader can find a complete proof in [4]). The first operation prescribes that as soon C becomes true the operation *must* be applied if $\neg T$ holds in order to reach the final state T. The second operation *may* be applied when C does not hold if the precondition T is true, making the postcondition $\neg T$ true.

Once all the operations were derived the were assigned to the agent responsible for the goal operationalized by those operations.

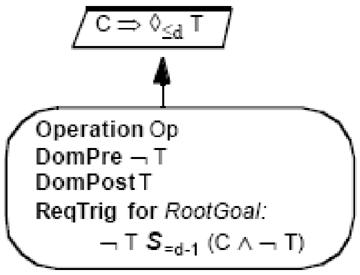

Fig. 3. Bounded achieve operationalization pattern

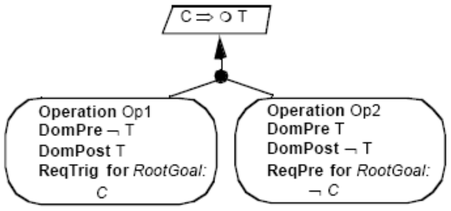

Fig. 4. Immediate achieve operationalization pattern

3 Architecture derivations

3.1 Method 1: Axel van Lamsweerde

This method [6] prescribes the use of three different steps: abstract a dataflow architecture from the KAOS specifications; derive and refine the dataflow using styles to meet architecturals constraints; refine the resulting architecture using design patterns to achieve non-functional requirements.

Step 1: Abstract a dataflow architecture The initial architecture is obtained from data dependencies between the different agents. The agents become software components while the data dependencies are modelled via dataflow connectors. The procedure followed is divided into two sub-steps.

1. Each agent that assumes the responsibility of a goal assigned to the software-to-be becomes a software component together with its operations.
2. For each pair of components C1 and C2, create a dataflow connector between C1 and C2 if

$$DataFlow(d, C1, C2) \Leftrightarrow Controls(C1, d) \wedge Monitors(C2, d) \tag{10}$$

One can note certain features. Due to the fact that the `COMM` agent does not control any variables no arrow comes from it. In fact `COMM` carries all the data among the different components but does not do any modifications. Moreover there is a dataflow connector between `PRECON` and `ALARM` while the real dataflow goes through `COMM`. This situation also happen between `Sensor` and `Precon`. The real dataflow passes through `DB` but there is no dataflow derived.

We believe that the underlying cause is the presence of low-level agents – `DB` and `COMM` – performing low-level functionalities – storage and transmission of data respectively – in the requirements. They were however needed to achieve certain goals. It resulted in a rather strange architecture.

Step 2: Style-based architectural refinement to meet architectural constraints In this step, the architectural draft obtained from step 1 is refined by imposing a "suitable" style, that is, a style whose underlying goals matched the architectural constraints. The main architectural constraint of our system [1, 2] is that all the components should be distributed. In fact, in the real system, only `PRECON` had to be built and integrated in to a pre-existing architecture characterized by centralized communications and by distributed components.

The only transformation rule mentionned in [6] did not match our architectectural constraints so we had to design a new one on the basis of what was needed. The resulting transformation rule is shown in Fig. 5.

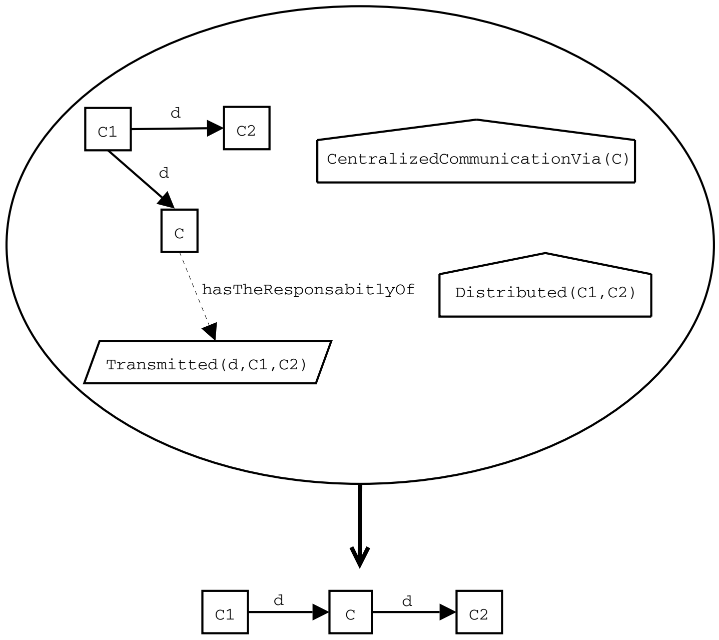

Fig. 5. Centralized communication architectural style

Once applied to the architecture every single communication is achieved in a centralized way through the communication module. The architectural constraints are now met.

Step 3: Pattern-based architecture refinement to achieve non-functional requirements The purpose of this last step is to refine further the architecture to achieve the non-functionnal requirements. These non-functional requirements (NFGs) can belong to two different categories: they are either quality-of-service or development goals. Quality-of-service goals include, among others, security, accuracy and usability. Development goals encompass desirable qualities of software such as *MinimumCoupling*, *MaximumCohesion* and *reusabilty*.

This step refines the architecture in a more local way than the previous one. Patterns are used instead of styles. The procedure is divided further into two intermediary steps.

1. For each NFG G, identify all the connectors and components G may constrain and, if necessary, instantiate G to those connectors and constraints.
2. Apply the refinement pattern matching the NFG to the constrained components. If more than one is applicable, select one using some qualitative technique (e.g., NFG prioritization).

Two refinement patterns were used on our system. The first is presented in Fig. 6. We wanted to have fault-tolerant communication between `PRECON` and `ALARM` because it is the core of the system. The most critical functions (i.e., the fault detection and the alarm management) are performed in these two component. That's why we wanted to make these modules as resistant as possible to any kinds of failure. One could note that the pattern was not applied exactly like it is defined in Fig. 6. The presence of the component `COMM` between `PRECON` and `ALARM` was however ignored because we believed it had no influence on the capacity of the pattern to achieve its goal.

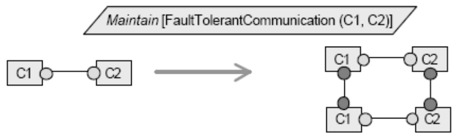

Fig. 6. Fault-tolerant refinement pattern

The second refinement pattern is shown in Fig. 7. It was introduced because both `Sensor` and `Management Unit` access and modify the same data – `SensorInformation`. We wanted to make sure that all the modifications made from both sides are consistent.

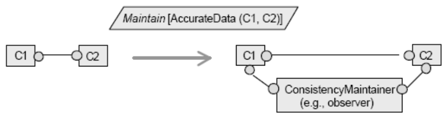

Fig. 7. Consistency maintainer refinement pattern

3.2 Method 2: Perry and Brandozzi

This method converts the goal oriented requirement specifications of KAOS into architectural prescriptions [7–9].

The components in an architecture prescription can be of three different types - process, data or connector. Processing components perform transformations on the data components. Data components contain necessary information for processing. The connector components, which can be implemented by data and/or processing components, provide mechanisms for component interactions. All components are characterized by goals that they are responsible for. The interactions and restrictions of these components characterize the system. The following is a sample component -

Component PRECON
Type Processing
Constraints FaultDetected
  RemedyActionSuggested
  PeriodicalChecksPerformed&ReportWritten
Composed of FaultDetectionEngine
  FaultInformation
  FaultDiagnosis
  SensorInformation
  SensorConnect
Uses /

This example shows a component called PRECON. *Type* denotes that the component is a processing component. The constraints are the various goals realized by PRECON. *Composed of* defines the subcomponents that implement PRECON in the next refinement layer. The last attribute *Uses*, indicates the components interacted with and the connectors used for their interactions.

There are well defined steps to go from KAOS entities to APL entities. The following table illustrates possible derivations.

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

In this method we create a component refinement tree for the architecture prescription from the goal refinement tree of KAOS. This is a three step process and may be iterated.

Step 1 In the first step we derive the basic prescription from the root goal of the system and the knowledge of the other systems that it has to interact with. In this case the software system is responsible for monitoring the power plant. Thus the root goal is assigned to the processing component "PowerPlantMonitoringSystem".

This goal is then refined into PRECON, ALARM, DataBase and Communication components. These refinements are obtained by selecting a specific level of the goal refinement tree. If we only take the root of the goal refinement tree, the prescription would end up being too vague. On the other hand if we pick the leaves, we may end up with a prescription that is too constrained. Therefore we pick a certain level of the tree which we feel allows us to create a very well defined prescription while avoiding a specification that overly constrains the lower level designs.

Step 2 Once the basic architecture is in place, we obtain potential sub components of the basic architecture. These are obtained from the objects in KAOS specification. We derive data, processing and connector components that can implement PRECON, ALARM, DataBase and Communication components. If in the third step we don't assign any constraints to these components, they are removed from the system's prescription.

The following are Preskriptor specifications of some candidate objects from the requirement specifications.

Component Fault
Type Data
Constraints ...
Composed of ...

Component FaultInformation
Type Data
Constraints ...
Composed of ...

Component SensorConnect

Type Connector
Constraints ...
Composed of ...

Component QueryManager
Type Processing
Constraints ...
Composed of ...

Since all the components derived from the KAOS' specification are data, we need to define various processing and connector components at this stage. At the next step we decide which of these components would be a part of the final prescription.

Step 3 In this step we determine which of the sub goals are achieved by the system and assign them to the previously defined components. With the goal refinement tree as our reference, we decide which of the potential components of step two would take responsibilities for the various goals. Note that this is a design decision made by the architect based on the way he chooses to realize the system. The components with no constraints are discarded, and we end up with the first complete prescription of the system.

Components like Fault were discarded from the prescription because they were not necessary to achieve the sub goals of the system. Instead of the Fault component we chose to keep FaultInformation. Different architects may make different decisions.

It is interesting to note that in our first iteration of the prescription Communication was a leaf connector with no subcomponents. It was responsible for realizing the necessary communication of the system. However the power plant communication was not uniform throughout the system. Different goals had different time, connection and security constraints for communication. In our first iteration we assumed that Communication component could handle these varying types of requirements on it. However then we realized that replacing the Communication component by more narrowly focused components was a step that helped illustrate these differences. Therefore we created the components UpdateDBConnect, FaultDetectionEngineAlarmManagerConnect and QueryDBConnect. As the names suggest, each of these were responsible for the communication in different parts of the system. Therefore it was easier to illustrate the different time and security constraints needed for each of these.

The following are the prescriptions for the sub components

Component UpdateDBConnect
Type Connector
Constraints Secure
TimeConstraint = 2 s
Composed of /
Uses /

Component QueryDBConnect
Type Connector
Constraints TimeConstraint = 5 s
Composed of /
Uses /

Component FaultDetectionEngineAlarmManagerConnect
Type Connector
Constraints Fault Tolerant
Secure
TimeConstraint = 1 s
Composed of /
Uses /

Step 4: Achieving non-functional requirements An additional fourth step in the prescription design process focuses on the non-functional requirements. Goals like reusability, reliability etc can be achieved by refining the prescription. This step is iterated till all the non-domain goals are achieved.

For this system we introduced additional constraints on the Database and the connector between Alarm and Precon (FaultDetectionEngineAlarmManager-Connect). In the case of Database an additional copy of the Database was introduced to ensure fault tolerance. With the introduction of a copy additional issues arose. For example, we needed to ensure that if the main database recovers from a failure, all the changes made on the second database since the failure should now be made on the main database. Once that's done the control should be shifted to the main database. This and several other additional constraints were thus defined.

As a second step, we also defined two copies of Alarm and Precon. This again created additional constraints. For example, each time one copy of Precon fails, the other one should take over without affecting the functioning of Alarm.

Other constraints to be considered include no data lost, sequence preserved, data transmitted in x time, mediation, transformation, coordination, hardware interaction, software interaction, human interaction, interoperability, security, fault tolerance, consistency, recovery, post recovery, retrieval of information, update of information etc.

Step 5: Box diagram Once the architecture was created we added a box diagram illustrating the various components and connectors. The component tree created as a result of the three steps did not show how the various components are linked through the connectors. The box diagram helps in visualizing this and thus gives a more complete view of the architecture.

4 Problems and Issues

There were some issues common to both architectures. First neither architecture has means of addressing fault tolerance, reliability etc as architectural constraints. The architectures are derived only from the goal oriented requirements, and there is a possibility that for some cases fault tolerance etc may be introduced for architectural reasons. Neither method has a well defined way of dealing with this. Secondly, we often had to work with inadequate information on the functioning of the power plant. We were unable to find any information on certain requirements like performance. Therefore performance was not included. However in a real world power plant system performance is an extremely critical issue.

4.1 Architecture 1

Step 1 proceeded well in generating a useful data flow architecture. However, in Step 2 where architectural styles are applied, there were only a few sample styles to look at. The power plant architecture was relatively small and we were unable to apply many of these styles to the architecture.

The third step requires the use of patterns to achieve non-functional requirements. There were various sample patterns given, however the small size of the power plant architecture limited the choice of patterns to apply. In some cases the patterns were not well documented so it was difficult to understand their application. On the other hand there were cases where it was required to apply two or more patterns to the same components. It was difficult to decide how to combine the patterns to realize this.

Another issue with the architecture was the creation of new components during the course of the derivation that had no operations. We also had to create some new connectors that did not have a complete definition.

Fig 8 and fig 6 show how to apply patterns to achieve interoperability and fault tolerance between components. However it is difficult to see how the patterns would be applied if components C1 and C2 needed to achieve both interoperability and fault tolerance. Another consideration is the order in which we apply these patterns to achieve a combination matters. There were no clear guidelines.

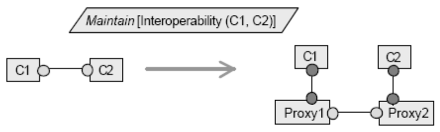

Fig. 8. Interoperability refinement pattern

We were unable to find suitable patterns for some other non functional requirements. For example, the power plant architecture required certain time constraints on different functions, but there were not suitable patterns to incorporate these time constraints with the architecture.

To achieve fault tolerance some components were replicated as illustrated in the pattern. It was difficult to determine which and how many components should be replicated. There wasn't enough information available on the functioning of the power plant to assign higher priority to some components and lower to others. The final decision was made based on the limited information provided.

An additional problem was illustrating the need to ensure consistency between the two replicated components. The communication between the components would change with the introduction of replicated components; however, it was difficult to explain how.

The alarm component was replicated since it was critical to ensure smooth functioning of the power plant. However we could not define the method of communication between the two copies of alarm, nor the method used to ensure consistency. It was also difficult to determine how the communication between Alarm-Operator and Alarm-Communication would change with the presence of an additional component and how this would change the current connector.

We could not determine the need for interoperability due to the lack of detailed system information.

The final architecture we obtained used a communication component to facilitate all communication for the system. However the communication between components often had different features and constraints. There were hardware connections, software connections, redundant components, different time constraints and different reliability constraints. It was not possible to incorporate these differences in communication in the architecture. One possibility discussed was to define communication as a connector instead of a component.

4.2 Architecture 2

Our first hurdle was the very first step. The architect is given a large degree of freedom in choosing an initial overall structure. While this may be appropriate for an experienced architect, it was difficult for us to determine how to start and how much to try to do in the first step. It was also difficult to realize how much leeway was allowed for each of the steps. We were unable to find sufficient guidance on the various steps in the process. There were no examples where we could find both the complete goal tree and the complete component tree. This would have allowed us to compare the trees and understand better the progression required to create the architecture. Some of the questions were

- What decisions regarding the architecture are made at step 1. Do we simply assign a root goal or do we need to anticipate the next steps and have a basic structure thought out?
- Is it possible to have refinement where the tree had more than three levels?
- If all the sub goals (of a root goal) are realized by a component, does the root goal (for those sub goals) still need to be assigned to a component?
- Ideally in the second step KAOS objects are used to create sub components. Was it possible to use agents in this step also? Sensor Management Unit was

an agent that we thought could be made a sub-component. However finally we used SensorInformation (which was an object) instead.
– Is it possible for a goal (and thus constraints) to be shared between sub components

Once the architecture was created we also added a box diagram illustrating the various components and connectors. The component tree created as a result of the three steps did not show how the various components are linked through the connectors. The box diagram helped in visualizing this and thus gave a more complete view of the architecture.

Once we obtained the component tree and the box diagram it provided us with different views. The tree seemed to indicate a hierarchy whereas the actual structure is quite different. The box diagram helped us realize the architecture as a network. Therefore there were different views of the system and structure based on the way we chose to look at it.

Additionally there were some components in the architecture that had no connectors. For example the AlarmInformation component under Alarm is a data component with various constraints on it, however it did not have a connector.

In the component tree and the resulting architecture there is no way to tell the data that is being passed through a connector. This made the architecture more difficult to understand. This information is particularly critical to describing the connectors. An alternative discussed for this problem was the possibility of having data as a constraint for a connector.

We also considered ways to explore the richness of connectors. Connectors can have different responsibilities like mediation, transformation and coordination. This richness would lead to a better design if we could portray this in the architecture prescription.

5 Comparison between the two methods

The most significant difference is that the first architecture is more low level. The components are described together with the operations that they have to perform creating a more rigid design. The second method uses an architecture prescription language which tends to be more high level. This allows the designer to pick a better solution at a low level. However at the same time it provides less guidance in getting to the solution.

The first method provides a more 'network type' view showing the various relationships and interactions between the components. The second method resulted in a component tree which was more hierarchical in nature. We needed an additional box diagram to better explain the component interaction. However both views though different were useful.

The first method was more systematic in the beginning. There was a clearly laid out approach for going from requirements to an architecture. The initial steps were simple enough to consider the possibility of automation in the future. However in the second method one of our biggest hurdles was getting past the first step. It was difficult to determine the basic composition with which to start. This was probably due to the high level nature of this method.

As we continued with the architecture derivation the first method got a little more confusing. We had problems choosing the appropriate patterns, and applying combinations of patterns. There was inadequate documentation on them to help in the process. On the other hand the second method became more manageable once we decided on an initial structure.

An interesting difference was that in the first method there were no constraints on the various connectors. Instead the focus was on the data that is passed through those connectors. In the second method we were able to specify various constraints for each of the connector, but there was no way of specifying the data that is passed through. In both cases we were unable to specify the differences possible in the nature of various types of connectors. For example, connectors for fault tolerant components may provide mediation. There was no way to specify this in either case.

With respect to non-functional requirements, in the first method we applied them by choosing the appropriate pattern. However in the second method we created additional constraints on the components to realize the non-functional requirements.

The second method takes as input the requirement specifications in KAOS and provides as output an architecture prescription. Obtaining a architecture prescription was a challenging process. There were several points where we were unclear on how to proceed. Therefore some suggestions are proposed in this section to make the various derivation steps easier to follow. The biggest problem encountered was with the very first step. It was difficult to determine how much of the architecture needs to be in place when deciding the first step. We did not know how to pick the components to determine the root and the second level of the component tree.

One way of approaching this is that the root goal of the component tree is simply the name of the system that is being implemented. In order to determine the second level of this tree we look at the second level of the goal tree. This gives a good idea of some of the high level goals of the system. We also look at some of the main subsystems that the given system would need to interact with in order to realize these goals.

The next step is to determine how detailed we want the second level of the component tree to be. We can choose to keep the second step simple which would typically include basic manager type components and a main connector component. These components are further spilt into detailed subsystems later.

In the Power plant problem, the subsystems that the main system interacts with are used to determine the second level components. This makes the second level of the tree more detailed. In this case - Precon, Alarm and Databases are the major subsystems that the power plant interacts with so these form the second level of the component tree. A communication component is also present to ensure proper communication between these various subsystems. The agents in the goal model are a way to start looking for the various subsystems involved. In both cases we looked at agents that are subsystems not agents that are people.

It is important to note that in both processes there is always a connector element present at the second level

Once the basic tree is in place the remaining steps are easy to follow.

The next problem faced was that the architecture specifies the various connectors in the subsystem. We can specify the constraints on these connectors. However there is no way to specify the data being passed through them. Various components do specify the connectors they use however information regarding the data being passed is absent under the connector description. A data flow model for this method would be useful in this. Another possibility is specifying data as a constraint for various connectors. Data along with the constraints would form a complete connector description

Once the component tree was in place it was felt that there was still a missing element to understand the architecture completely. The component tree gave us a hierarchical type view of the system; but that was not adequate so we added a box diagram to give us a network type view. This is essential in understanding how the system worked. This diagram also helped in understanding the connectors of the system because it told us the way these connectors linked to components. This thereby helped in getting an understanding of the data that would be passed through these connectors. Understanding of the data passed is essential to getting a complete description of the connectors.

6 Conclusions

In this research we took a real world example of a power plant system and systematically obtained goal-oriented requirement specifications. We then created two architectures that satisfy the requirements. We analyzed and compared the results. Both architectures provided us with different but nonetheless useful views of the system. We used our example to create further well defined derivation methods making this critical step of the system design process easier.

Subsequently, this case study became the foundation for two masters theses: Jani explored how styles and patterns provide some non-functional constraints such as reliability and fault tolerance in the Perry/Brandozzi approach [12]; and Vanderveken investigated adding a behavioral view to van Lamsweerde's KAOS methods and precisely describing and applying transformation patterns. [13].

A good start, but much further work still needs to be done.

References

1. Coen-Porisini, A., Mandrioli, D.: Using trio for designing a corba-based application. Concurrency: Practical and Experience **12** (2000) 981–1015
2. Coen-Porisini, A., Pradella, M., Rossi, M., Mandrioli, D.: A formal approach for designing corba based applications. In: ICSE 2000 - 22nd International Conference on on Software Engineering, Limerick, ACM Press (2000) 188–197

3. Massonet, Ph., van Lamsweerde, A.: Formal refinement patterns for goal-driven requirements elaboration. In: FSE-4 - 4th ACM Symposium on the Foundations of Sofware Engineering, San Fransisco, ACM Press (1996) 179–190
4. Letier, E., van Lamsweerde, A.: Deriving operational software specifications from system goals. In: FSE-10 - 10th ACM Symposium on the Foundations of Sofware Engineering, Charleston, ACM Press (2002) 119–128
5. Letier, E., van Lamsweerde, A.: Agent-based tactics for goal-oriented requirements elaboration. In: ICSE 2002 - 24th International Conference of Sofware Engineering, Orlando, ACM Press (2002) 83–93
6. van Lamsweerde, A.: From system goals to software architecture. In Bernardo, M., Inverardi, P., eds.: Formal Methods for Software Architectures. Volume 2804 of Lecture Notes in Computer Science. Springer-Verlag (2003) 25–43
7. Brandozzi, M., Perry, D.E.: Transforming goal oriented requirement specifications into architectural prescriptions. In Castro, Kramer, eds.: STRAW 2001 - From Software Requirements to Architectures. (2001) 54–60
8. Brandozzi, M., Perry, D.E.: Architectural prescriptions for dependable systems. In: ICSE 2002 - International Workshop on Architecting Dependable Systems, Orlando (2002)
9. Brandozzi, M.: From goal oriented requirements specifications to architectural prescriptions. Master's thesis, The University of Texas at Austin (2001)
10. Jani, D., Vanderveken, D., Perry, D.: Experience report deriving architectural specification from kaos specification. Technical report (2004) Also avaiable at http://www.ece.utexas.edu/∼perry/papers/R2A-ER.pdf.
11. Manna, Z., Pnueli, A.: 3. In: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag (1992)
12. Jani, D.: Deriving architecture specifications from goal oriented requirement specifications. Master's thesis, The University of Texas at Austin (2004)
13. Vanderveken, D.: Deriving architecture descriptions from goal oriented requirements. Master's thesis, University of Louvain, Belgium (2004)