Copyright
by
Divya Jani
2004

Deriving Architecture Specifications from Goal Oriented Requirement Specifications

by

Divya Jani, B.S.

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

May 2004

Deriving Architecture Specifications from Goal Oriented Requirement Specifications

APPROVED BY

SUPERVISING COMMITTEE:

____

Dewayne Perry, Supervisor

____

Suzanne K Barber

This thesis is dedicated to my grandmother for her unconditional love and support.

Acknowledgments

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

To my supervisor, Professor Dewayne Perry, for his constant guidance and encouragement through this thesis work and for being a sincere friend and mentor.

To Damien Vanderveken, in whom I found not just a great colleague but a great friend.

Thanks to Professor Suzanne Barber, for all that I learnt in the classes I took with her.

I am also very grateful to Umesh for all his help with the LaTeX.

To my family, for supporting me throughout my education.

To Roshn, for always being there for me.

And to my two year old nephew Raghav for teaching me to never give up.

Deriving Architecture Specifications from Goal Oriented Requirement Specifications

Divya Jani, M.S.E
The University of Texas at Austin, 2004

Supervisor: Dewayne Perry

The first step in realizing a software system is collecting the requirements and using them to obtain an architecture. In this thesis we use an example of a power plant system and obtain a goal oriented requirement specification for it. The KAOS requirement specification language was used for this. Next we use two different methods to arrive at an architecture. These methods are compared and contrasted. Some of the problems encountered in the derivation process are specified. Based on these problems several improvements are suggested for method two. We then look at styles and patterns to achieve non-functional requirements.

Table of Contents

Acknowledgments **v**

Abstract **vi**

List of Figures **x**

Chapter 1. Introduction **1**

Chapter 2. Requirements derivation using the KAOS method **3**

2.1 Goal Model 3

2.1.1 Goal model elaboration 3

2.1.2 Goal model characteristics 4

2.2 Object Model 8

2.2.1 Object Model Elicitation 8

2.2.2 Object Model Characteristics 9

2.3 Agent Model 10

2.3.1 Agent Model Elaboration 10

2.3.2 Agent Model characteristics 11

2.4 Operation Model 12

2.4.1 Operation Model Elaboration 12

2.4.2 Operation Model Characteristics 14

Chapter 3. Architecture derivation **17**

3.1 First method: Axel van Lamsweerde 17

3.1.1 Step 1: From software specifications to abstract dataflow architectures 17

3.1.2 Step 2: Style-based architectural refinement to meet architectural constraints 19

3.1.3 Step 3: Pattern-based architecture refinement to achieve non-functional requirements 19

3.2 Second method: Dewayne Perry and Manuel Brandozzi 23
3.2.1 First step 24
3.2.2 Second step 25
3.2.3 Third step 27
3.2.4 Achieving non-functional requirements 29
3.2.5 Box diagram 30
3.3 Problems and Issues 30
3.3.1 Architecture 1 31
3.3.2 Architecture 2 35
3.4 Comparison between the two methods 37

Chapter 4. Suggested modifications to the second method: Perry and Brandozzi 40

Chapter 5. Styles and Patterns 43
5.1 Introduction to styles and patterns 43
5.1.1 Model View Controller Pattern 44
5.1.2 Whole Part Pattern 45
5.1.3 Proxy Pattern 46
5.1.4 Publisher Subscriber Pattern 48
5.1.5 Master and Slave Pattern 50
5.1.6 Achieving Fault Tolerance 51
5.1.7 Unit of Work Pattern 53
5.1.8 Memento pattern 55
5.1.9 Additional Transformations 56

Chapter 6. Conclusion 59

Appendices 61
.1 Goal specifications 62
.2 Object Specifications 83
.3 Agents Specifications 91
.4 Operations specifications 99
.5 Axel van Lamsweerde Architecture 112

.6 Architecture Prescriptions 116
.7 Additional constraints on the system 125
.7.1 Constraints on the Database 125
.7.2 Constraints on the connector between ALARM & PRECON (i.e., FaultDetectionEngineAlarmManagerConnect) 126
.8 Goal Oriented Requirements to Architecture Prescription - Updated 128

Bibliography **137**

Vita **140**

List of Figures

2.1 Milestone refinement pattern 4
2.2 Communication refinement subtree 6
2.3 Bounded achieve operationalization pattern 13
2.4 Immediate achieve operationalization pattern 13

3.1 Centralized communication architectural style 20
3.2 Fault-tolerant refinement pattern 22
3.3 Consistency maintainer refinement pattern 22
3.4 Interoperability refinement pattern 33
3.5 Fault-tolerant refinement pattern 33

5.1 Model view controller pattern 45
5.2 Whole part pattern 46
5.3 Proxy pattern 48
5.4 Publisher subscriber pattern 49
5.5 Master slave pattern 51
5.6 Fault Tolerance 52
5.7 Unit of work pattern 54
5.8 Memento pattern 56
5.9 Additional transformations pattern 57

1 Goal diagram 63
2 Goal diagram continued 64
3 Object diagram 84
4 Agent diagram 92
5 Agent diagram continued 93
6 Step 1: dataflow architecture 113
7 Step 2: style-based refined architecture 114

8 Step 3: pattern-based refined architecture 115
9 Component refinment tree 117
10 Box diagram of the architecture 124
11 Goal refinement tree for the paper selection process 130
12 Component refinement tree for the paper selection process . . 131

Chapter 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 are transformed to an architecture that can be understood by developers. There are several different ways to derive an architecture and two of those ways are explored here.

The system we used throughout this report was an example of a power plant that was obtained from [5, 6]. Our first step was to create a goal-oriented requirements specification from the information available. The KAOS requirement specification language is used [8, 9, 11]. The power plant description was not complete so 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 [14]. The various steps are explained in detail in one of the following sections of this report. We have also described some of the problems encountered during the derivation process.

The second method results from the work of Dewayne Perry and Manuel Brandozzi (University of Texas at Austin). Their work is presented in [1–3]. The resulting architecture and some of the derivation issues are described

in this report. 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.

Chapter 2

Requirements derivation using the KAOS method

2.1 Goal Model

2.1.1 Goal model elaboration

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 [5]. 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 our engineering skills, on Professor Perry's advice and on our common sense in order to gather requirements that are as realistic as possible.

The following steps were followed in order to build the goal model. First of all, the informal definition of goals that are mentioned in [5] were carefully written down. From that, a first goal refinement tree was built. This first draft was all but complete. This tree was completed thanks to a refinement/abstraction process. The version we obtained at that point was still totally informal. Temporal first-order logic [10] was then used to remove this weakness. It enabled us to ensure our refinement tree was correct, complete and coherent. The use of refinement patterns as described in [11] served as

a guidance. The milestone-driven pattern in particular was applied numerous times. It prescribes that some milestone states are mandatory in order to reach the final one. This pattern is presented in fig 2.1. 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.

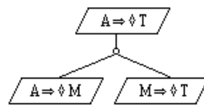

Figure 2.1: Milestone refinement pattern

Because of the iterative nature of the requirements gathering process, the goal model underwent subsequent changes. The reasons for that were various e.g., coherence between the different models forming the KAOS specifications, enhancements, simplifications,etc. A complete goal description can be found in Appendix .1.

2.1.2 Goal model characteristics

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 *AlarmCorrecltyManaged*. 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 – the most widely used – the following example will be developed. Let's consider the goal *AlarmRaisedIfFaultDetected* with its formal definition

$$\big(\forall f : Fault, \exists! l : Location, \exists! a : Alarm\big)\big(Detected(f, l) \Rightarrow \Diamond Raise(f, a)\big) \tag{2.1}$$

This goal is refined using the milestone refinement pattern presented in fig 2.1 by instanciating the parameters as follows:

$$A : \big(\forall f : Fault, \exists! l : Location\big)\big(Detected(f, l)\big) \tag{2.2}$$

$$M : \big(\exists fi : FaultInformation\big)\big(f \equiv fi \wedge Transmitted(fi, PRECON, ALARM)\big) \tag{2.3}$$

$$T : \big(\forall fi : FaultInformation, \exists! a : Alarm\big)\big(Raised(fi, a)\big) \tag{2.4}$$

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., the raising of the alarm.

In order to have a system as robust as possible various goals have been 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 *DataCorrectlyUpdata* – 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*). The communication has also been constrained in order to prevent any transmission problems.

The refinement of the goal *DataTransmittedToDB* is the result of that policy. The goal was refined as shown in Fig. 2.2

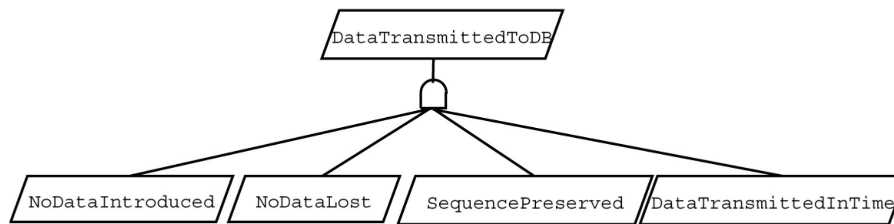

Figure 2.2: Communication refinement subtree

The three first subgoals ensure the corectness 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 [1]:

[1]X stands for *SensorInformation*, *FaultInformation*, *AlarmInformation*, *FaultDiagnosis*

$$NoDataIntroduced \quad :$$

$$\big(\forall x : Data\big)\big(Transmitted(X, _, _) \wedge x \in Transmitted(_) \Rightarrow x \in X\big) \qquad (2.5)$$

$$NoDataLost \quad :$$

$$\big(\forall x : Data\big)\big(x \in X \wedge Transmitted(X, _, _) \Rightarrow x \in Transmitted(_)\big) \qquad (2.6)$$

$$SequencePreserved \quad :$$

$$\big(\forall x, y : Data, \exists u, v : Data\big)\big(x, y \in X \wedge Transmitted(X, _, _)$$
$$\wedge Before(x, y, X) \Rightarrow u, v \in Transmitted(X) \wedge$$
$$Before(u, v, Transmitted(X)) \wedge x = u \wedge y = v\big) \qquad (2.7)$$

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:

$$DataTransmittedWithinTimeConstraint \quad :$$
$$\neg Transmitted(fi, PRECON, ALARM) \qquad \Rightarrow \Diamond_{\leq 1s}$$
$$Transmitted(fi, PRECON, ALARM) \qquad (2.8)$$

and *AlarmDiagnosis*

2.2 Object Model

2.2.1 Object Model Elicitation

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 in order 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 *SanityCheckPerformed*.

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:

$$Activated = true \Rightarrow DeactivationTime = null \tag{2.9}$$

The purpose of certain attributes is to prepare for change. The reconfiguration function was finally not taken into account in the elaboration of the different 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 in order 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 of 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 goal model can be found in Appendix .2.

2.2.2 Object Model Characteristics

The main characteristic of the model is the presence of two different levels of representations for the concepts *Sensor*, *Fault* and *Alarm*. The first level refers to the object in 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 concepts – 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 rela-

tionship *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 reason for that is the resulting simplicity and the easiness of traceability.

2.3 Agent Model

2.3.1 Agent Model Elaboration

The definition of the agents was extracted mostly from [5, 6]. We drew inspiration from the existing agents. Each leaf goal from the Goal Model was assigned to an agent. We made sure that every agent has the capacity to assume the responsibility of that goal. By capacity we mean that every agent could monitor or control, depending on the case, every single variable appearing in the formal definition of a goal the agent has to ensure. For further details please refer to [8].

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

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

A complete agent model can be found in Appendix .3.

2.3.2 Agent Model characteristics

As it was already said, most of the agents come from the existing system. This is the case for `PRECON`, `ALARM`, `COMM`, `DB` and `Sensor`. The names used in [5] may be different but basically the functions performed are the same.

`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 request 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 can be either part of the sofware-to-be or part of the environment. For example, `PRECON` belongs to the first class while `Sensor` belongs to the second one. This distinction in agents results also in a goal differentiation. In fact the goals assigned to environment agents are expectations while the others are requirements. This leads 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 those kind of tests should not be the responsibitlity

of the `Sensor` from a conceptual point of view.

2.4 Operation Model

2.4.1 Operation Model Elaboration

The operation model was the the last one to be constructed because it relies on a precise formal definition of the goals in order to be derived automatically. 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 did an extensive use of the operationalization patterns described in [9] in order to derive complete operation specifications. It enabled us to save a lot of time on proofs. It is even more true than for the goal refinement pattern because we found the application of the operationalization very systematic.

Two patterns were particularly useful and we used them numerous times. The first one is the bounded achieve pattern described in Fig. 2.3. Its applicabilty condition (i.e., $C \Rightarrow \Diamond_{\leq d} T$) makes it very popular. In fact most of our system's goals have that form. The operation specification prescribes that $\neg T$ becomes T as soon as $C \land \neg T$ holds for $d-1$ time units. It is then straightforward to see that such a specification operationalizes the goal $C \Rightarrow \Diamond_{\leq d} T$.

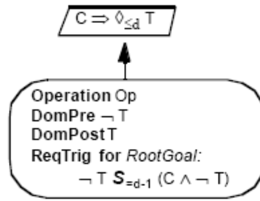

Figure 2.3: Bounded achieve operationalization pattern

The second most useful pattern was the immediate achieve pattern described in Fig. 2.4. 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. We will give a short explanation why but the interested reader can find a complete proof in [9]. 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.

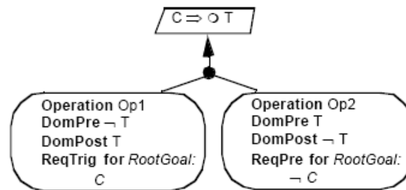

Figure 2.4: Immediate achieve operationalization pattern

Once all the operations were derived the were assigned to the agent responsible for the goal operationalized by those operations. Furthen details can be found in Appendix .4.

2.4.2 Operation Model Characteristics

We will present in this section an illustration of the two operationalization patterns mentionned in the previous section.

For the first pattern, we will examine the operationalization of the goal `FaultInformationTransmittedWhenFaultDetected`. Its formal defintion is given by

$$\begin{aligned}&\big(\forall f : Fault, \exists! l : Location, \exists! fi : FaultInformation\big)\\&\qquad\big(Detected(f,l) \wedge f.ID = fi.ID \Rightarrow \Diamond_{\leq 1s}\\&\qquad Transmitted(fi, PRECON, ALARM)\end{aligned} \tag{2.10}$$

We can instantiate the pattern presented in Fig 2.3 with the following parameters.

$$C \quad : \quad Detected(f,l) \wedge f.ID = fi.ID \tag{2.11}$$

$$T \quad : \quad Transmitted(fi, PRECON, ALARM) \tag{2.12}$$

The operation resulting from the application of the pattern is:

Operation TransmitFaultInformation

DomPre ¬ Transmitted(fi,PRECON,ALARM)

DomPost Transmitted(fi,PRECON,ALARM)

ReqTrig for FaultInformationTransmittedWhenFaultDetected

¬ Transmitted(fi,PRECON,ALARM) $\mathbf{S_{=1ms}}$ Detected(f,l)

∧ f.ID=fi.ID ∧ ¬ Transmitted(fi,PRECON,ALARM)

Note that as $d - 1$ time units is zero we simply took a smaller time unit.

To illustrate the second pattern consider the goal `SanityCheckPerformed` whose formal defintion is given by

$$\begin{aligned}(\forall s : Sensor)\big(&\neg s.workingProperly \wedge \\ &s.status =' on' \Rightarrow \circ s.status =' off'\big)\end{aligned} \tag{2.13}$$

The instantiation of the immediate achieve pattern presented in Fig. 2.4 is straightforward.

$$C \; : \; \neg s.workingProperly \wedge s.status =' on' \tag{2.14}$$

$$T \; : \; s.status =' off' \tag{2.15}$$

The first operation derived thanks to application of the pattern is

Operation SwitchSensorOff

DomPre s.status='on'

DomPost s.status='off'

ReqTrig for SanityCheckPerformed

¬ s.workingProperly

and the second one is

Operation SwitchSensorOn

DomPre s.status='off'

DomPost s.status='on'

ReqPre for SanityCheckPerformed

s.workingProperly

Chapter 3

Architecture derivation

3.1 First method: Axel van Lamsweerde

The architecture derived in this section will be derived using the method developed by Axel van Lamsweerde in [14]. His method prescribes the use of three different steps. The first step consists of the derivation of a abstract dataflow architecture from the KAOS specifications. This first draft is next refined using styles in order to meet architecturals constraints. The architecture obtained is finally refined using design patterns so as to achieve non-functional requirements. One section will be devoted to each step. After that the issues encountered will be discussed.

3.1.1 Step 1: From software specifications to abstract dataflow architectures

The first 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, drive a dataflow connector between C1 and C2 if

$$DataFlow(d, C1, C2) \Leftrightarrow Controls(C1, d) \wedge Monitors(C2, d) \quad (3.1)$$

This step is very systematic. The result is shown in Fig. 6.

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 results in a strange architecture. The result can be seen in Fig. 6 in Appendix .5.

3.1.2 Step 2: Style-based architectural refinement to meet architectural constraints

In this step, the architectural draft obtained from step 1 is refine by imposing a "suitable" style, that is, a style whose underlying goals match the architectural constraints. The main architectural constraint of our system [5], [6] is that all the components should be distributed. In fact, in the real system, only `PRECON` had to be built and it has to integrate in a pre-existing architecture characterized by centralized communications and by distributed components.

The only transformation rule mentionned in [14] did not match our architectectural constraints so we had to design a new one considering what we thought we should obtain. The resulting transformation rule is shown in Fig. 3.1.

This style was applied on our architectural draft and the result is shown in Fig. 7 in Appendix .5.

As you can see, the architecture looks now closer to what we expected. Every single communication is achieved in a centralized way through the communication module. The architectural constraints are now met.

3.1.3 Step 3: Pattern-based architecture refinement to achieve non-functional requirements

The purpose of this last step is to refine further the architecture in order to achieve the non-functionnal requirements. These non-functional re-

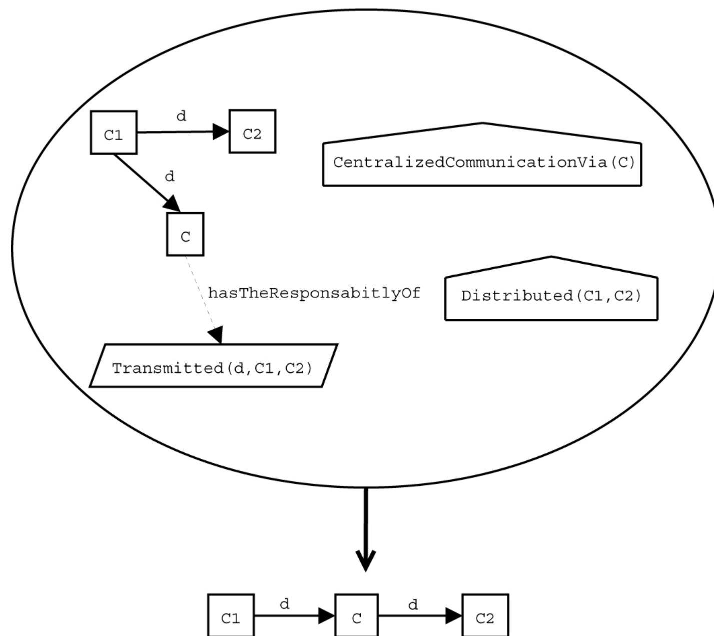

Figure 3.1: Centralized communication architectural style

quirements can belong to two different categories: they can be either quality-of-service or developemnt 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 to follow could be 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 one is presented in Fig. 3.5. We wanted to have a 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 managemnet) are performed in those two component. That's why we wanted to make those modules as resistant as possible to any kinds of failure. One could note than the pattern was not applied exactly like it is defined in Fig. 3.5. The presence of the component `COMM` between `PRECON` and `ALARM` was however ignored because

we believed it has no influence on the capacity of the pattern to achieve its goal.

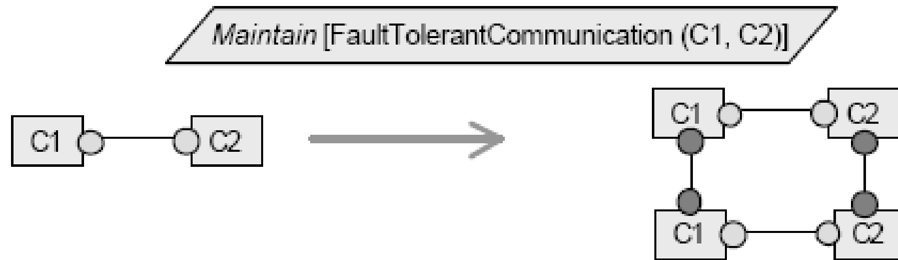

Figure 3.2: Fault-tolerant refinement pattern

The second refinement pattern we used is shown in Fig. 3.3. 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.

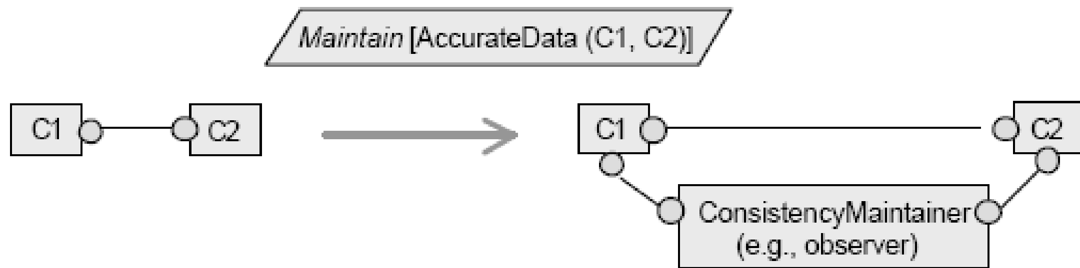

Figure 3.3: Consistency maintainer refinement pattern

The final architecture is presented in Fig. 8 in Appendix .5.

3.2 Second method: Dewayne Perry and Manuel Brandozzi

The second method converts the goal oriented requirement specifications of KAOS into architectural prescriptions.

The components in an architecture prescription can be of three different types - process, data or connector. Processing components perform transformation the data components. The data components contain the necessary information. The connector components, which can be implemented by data or processing components, hold the system together. 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. It thereby defines the constraints on the component. *Composed of* illustrates the sub components that implement PRECON in the next refinement layer. The last attribute *Uses*, indicated what are the components used by this component. It also specifies the connectors used for the interaction.

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

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.

3.2.1 First step

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 defined as "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 constrains the lower level designs.

3.2.2 Second step

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 won't be a part of 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 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.

3.2.3 Third step

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 of 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 creating sub components for Communication component was a step that helped illustrate these differences. Therefore we created the sub components - UpdateDBConnect, FaultDetectionEngineAlarmManagerConnect and Query-DBConnect. 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 /

A detailed prescription can be found in Appendix .6.

3.2.4 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 (FaultDetectionEngineAlarm-ManagerConnect).

In case of Database an additional copy of the Database was introduced to ensure fault tolerance. With the introduction of a copy additional issues arise. For example, we need 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 an 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.

A comprehensive list of additional constraints can be found in Appendix .7.

Other areas where constraints can be considered in the future 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.

3.2.5 Box diagram

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 helps in visualizing this and thus gives a more complete view of the architecture. The complete box diagram is shown in Fig.10 in Appendix .6.

3.3 Problems and Issues

The following section provides an overview of some of the problems encountered while working on the architecture.

There were some issues common to both architectures. Firstly 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 very critical to the functioning.

Next we describe the problems encountered specific to each architecture.

3.3.1 Architecture 1

Once the requirements are finalized, the first step is to obtain an abstract dataflow architecture. The dataflow architecture is obtained by using functional goals assigned to software agents. The agents become architectural components and then dataflow connectors are derived from input/output dependencies.

In the next stage architectural styles are applied. At this point 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.

An other 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.

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.

The following two figures 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 say components C1 and C2 needed to achieve both interoperability and fault tolerance. Another consideration would be if the order in which we apply these patterns to achieve a combination matters. There were no clear guidelines provided to realize this.

We were unable to find suitable patterns for some other non functional requirements. The power plant architecture required certain time constraints on different functions, however it was not possible to illustrate these time constraints with the architecture.

In order to achieve fault tolerance some components were made redun-

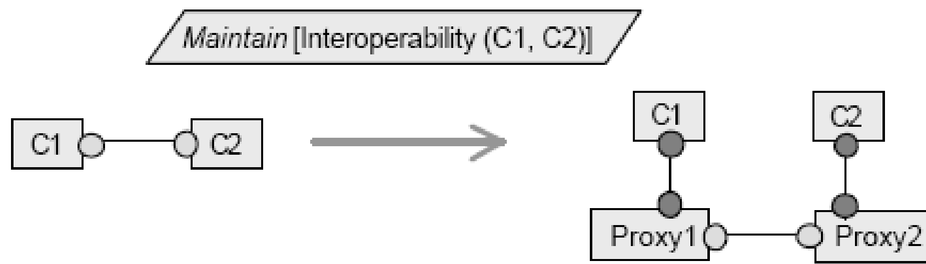

Figure 3.4: Interoperability refinement pattern

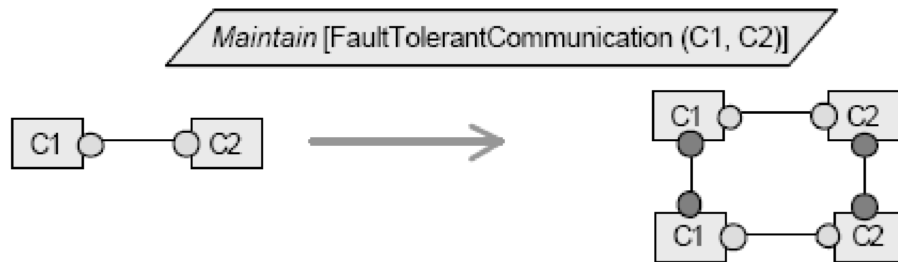

Figure 3.5: Fault-tolerant refinement pattern

dant as illustrated in the pattern. It was difficult to determine which and how many components should be redundant. 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 redundant components. The communication between the components would change with the introduction of redundant components however it was difficult to explain how.

The alarm component was made redundant 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, and 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 illustrate these differences in communication with the architecture. One possibility discussed was to define communication as a connector instead of a component.

3.3.2 Architecture 2

This method takes as input the requirement specifications in KAOS and provides as output an architecture specification in an architectural prescription language (APL) - Preskriptor. Creating the architecture is a three step process where in the first step the basic prescription is derived from the root goal for the system and the knowledge of the other systems it has to interact with. In the second step objects in the KAOS specification are used to derive components that are potential sub components of the basic architecture. In the third step an appropriate degree of refinement of the goal refinement tree is selected. At this point the sub goals that are achieved by the system are assigned to the sub components created in step two. This defines the basic architecture of the system. Further refinement can then be done to achieve various non functional properties. 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. Therefore our first hurdle was the very first step. It was difficult 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. Some of the questions that came up 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 helps in visualizing this and thus gives 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 seems to indicate a hierarchy whereas the actual structure is quite different. The box diagram helps 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 does 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 makes 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. It would lead to a better design if we could portray this in the architecture.

3.4 Comparison between the two methods

In this section we compare the two architecture derivation methods and the resulting architectures . 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 combination 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 design.

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. On the other hand, in the second method we were able to specify various constraints for each of the connector, however 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 fault tolerant components may have mediation type connectors. There was no way to specify this in either case.

As concerns 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.

Chapter 4

Suggested modifications to the second method: Perry and Brandozzi

The second method takes as input the requirement specifications in KAOS and provides as output an architecture specification in an architectural prescription language (APL) - Preskriptor. Obtaining the 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 sub systems 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.

A different approach is seen in the Power plant problem. In this case 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 case of the powerplant - 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 look 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 proceeding 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, however it was not adequate so we added a box diagram to give us a network type view. This is essential in understanding how the system works. This diagram also helps in understanding the connectors of the system because it tells us the way these connectors link to components. This thereby helps 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.

Chapter 5

Styles and Patterns

5.1 Introduction to styles and patterns

For method two (Brandozzi and Perry) we were able to create a final prescription following the three step process discussed. The fourth step of the prescription design process focuses on refining the prescription to achieve non functional requirement. Non functional requirements are a very important part of a system and it is essential to consider them as early as possible in the system design process. These goals can alter the prescription at a component level or affect the design of the whole system. Examples of these goals include reusability, fault tolerance, reliability etc. Taking into account non functional requirements while designing a prescription can have three kinds of effects on an already designed prescription of a system - it can lead to introduction of new components, transformation of the system topology or further constraining of already existing components.[3] Architecture styles and patterns can be used to provide solutions for realizing several of these goals. Styles and patterns usually deal with a specific problem in the design or implementation of software system and provide a solution to it. These come from practical, well proven design experience and can be used to solve design problems effectively. Styles and patterns determine the basic structure of the solution to a particular design

process [4]. This solution can then be implemented according to the basic needs of the system. This section looks at some specific styles and patterns used to achieve various non functional requirements for the power plant system prescription. It discusses the pattern or styles used, the transformations done and the non function requirements satisfied.

5.1.1 Model View Controller Pattern

The first transformation is derived from the Model-View-Controller architecture pattern [4]. The MVC architectural pattern divides an interactive application into three components. The model contains the core data and functionality. This is independent of specific output representations or input behavior. View components display information to the user by obtaining data from the model. There can be multiple views of a model. Controllers receive input, usually as events like mouse movement or keyboard input and translate them to service requests. The user interacts with the system solely through the controllers. In the power plant architecture the Database contains all the relevant data and query functionality. The InteractionManager is used to manage requests from the operator who is human. This pattern adds UserView which now handles the various views provided to the operator. The InteractionManager serves as the controller and translates user requests to queries on the database.

UserView is responsible for presenting information to the operator. Different views can be used to present information in different ways. Thus the

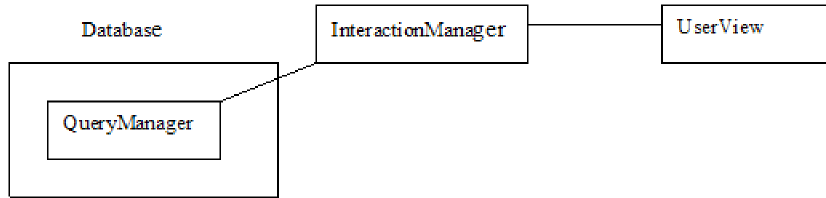

Figure 5.1: Model view controller pattern

user interface is not as tightly linked to the core data anymore allowing for more flexibility and choices. This allows for the independence of the 'look and feel' of the system and easier portability to new platforms.

5.1.2 Whole Part Pattern

This transformation is obtained from the idea of the whole-part design pattern[4]. The whole part design pattern helps with the aggregation of components that together form a semantic unit. A whole object represents a collection of smaller objects which are called parts. The object forms a semantic grouping of its parts in that it coordinates and organizes their collaboration. A common interface is provided to access the functionality of the parts. In this case we create a PRECON component that encapsulates other components - FaultInformation, FaultDetectionEngine, SensorInformation, FaultDiagnosis. These components are selected because they logically come under PRECON. An interface is also defined which allows for access to the functionality of the encapsulated components. Components are accessed

only through the interface, not directly. Essentially the transformation creates a subsystem.

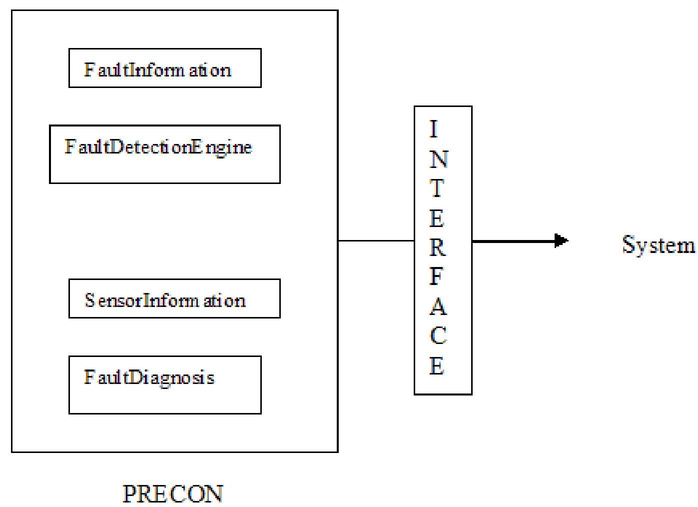

Figure 5.2: Whole part pattern

This can also be applied to the rest of the system to form Alarm and Database subsystems. This transformation allows for separation of concern. Each concern can be implemented by a separate part. Reusability of the various subsystems created is promoted. At the same time various components that form the subsystems can also be reused.

5.1.3 Proxy Pattern

Proxy[4] is a popular design pattern which allows the clients of a component to communicate with a representative rather than the component itself.

This representative is called a proxy and acts as the interface to the component. Direct access to a component may not always be the best approach in terms of efficiency and protection from unauthorized access etc. Proxy provides some of additional control methods that are needed such as access control checking. At the same time the access mechanism is kept simple.

This proxy pattern can have several variants. A protection proxy protects the component from unauthorized access. In order to realize this, the proxy checks the access rights in each instance. It is also possible to give various clients different access permissions. A cache proxy maintains a data area to temporarily hold results. In this case the more frequently desired results can be accessed quicker providing more efficient communication. A synchronization proxy controls multiple simultaneous client accesses and can also distinguish between read and write accesses. A virtual proxy assumes that an application references secondary storage and loads on parts of the application as needed and frees up space when parts are no longer needed. A firewall proxy protects local clients by checking outgoing requests and incoming for compliance with internal security policies. A specific proxy can play several of these roles and fulfill various responsibilities.

In the case of the power plant the database is accessed frequently for various queries. Various different types of queries can be placed but they fall under some specific categories and are repeated often. In this case a proxy component is created which now provides access to the QueryManger of the database to the InteractionManager component. The proxy also ensures a

safe and correct access by checking access rights thus providing security. The retrieval process can also be made more efficient by providing a cache.

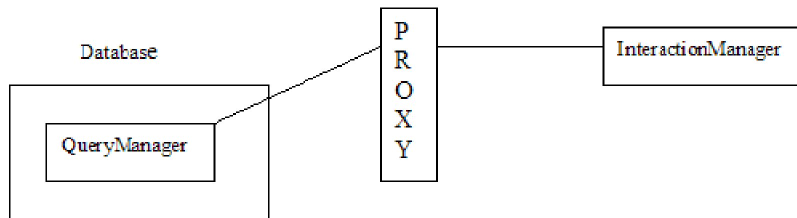

Figure 5.3: Proxy pattern

5.1.4 Publisher Subscriber Pattern

The following transformation is derived from the Publisher-Subscriber pattern[4]. This pattern helps to keep the state of cooperating components synchronized. One dedicated component takes the role of a publisher. Other components dependent on changes in the publisher are called its subscribers. The publisher maintains a list of current subscribers. Whenever the publisher changes state, it sends a notification to all its subscribers. The subscribers in turn retrieve the changed data. For the power plant system that database is an important component. In order to ensure reliability and prevent any loss of data two copies of this database are maintained. One of these copies is the main database whereas the other is a backup. It then becomes important to maintain consistency between the two copies. A pattern similar to publisher-subscriber can be used for this where the main database is the publisher and

the backup database is the subscriber. However the two database copies have several other constraints. If the main database fails the backup database should take over and update itself as needed. Once the main database recovers from the failure all the updates made to the backup database should be made to the main database. Therefore for this particular case sometimes changes may be made to the subscriber database and they should be replicated to the publisher database. However this would only happen in the case of failure and recovery.

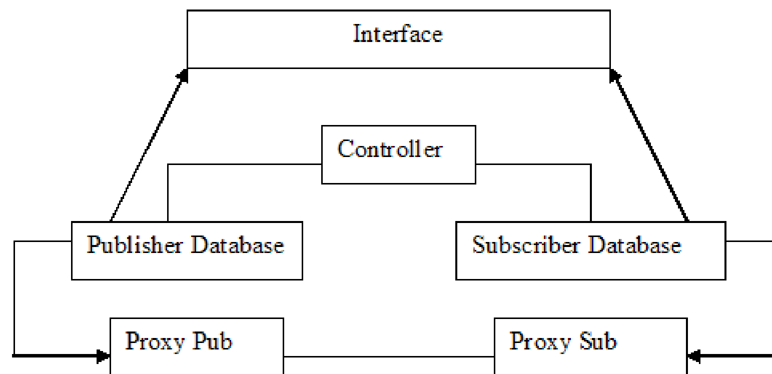

Figure 5.4: Publisher subscriber pattern

An additional controller is added to maintain these constraints. The controller handles control among the databases in case of failure and updates in case of recovery. The interface is connected to both databases and changes take place to the publisher database when all functions are running correctly and to the subscriber database in case of failure. The presence of two databases

increases reliability since one database can take over if the other fails ensuring that the data is always current and available.

5.1.5 Master and Slave Pattern

The master and slave design pattern[4] supports fault tolerance and computational accuracy. A master component delegates work to identical slave components and computes final results from the results the slaves return. Thus the work is partitioned into several sub tasks and assigned to slaves who perform the tasks independently and final result is put together from the various partial results obtained. This structure consists of one master and at least two slaves. The pattern can support fault tolerance. In that design the master simply delegates a certain task to a specific number of slaves thereby ensuring that the result is returned as long as at least one slave does not fail. However the master is the critical component and must stay alive to make the structure operate. Another application for this pattern is found in parallel computation. In this case the master divides a task into smaller subtasks which are performed independently by the slaves. The master then builds the final results. A third application is for computational accuracy. Here the execution of a service is delegated to at least three different slaves. Once the calculations by all the slaves are complete the master uses a voting system to handle inaccuracies. The master may select the result that is returned by the greatest number of slaves or take an average of all the results. The FaultDetectionEngine component of the architecture is responsible for performing calculations to check if a

fault is detected in the system. A correct detection of fault is very important for the power plant system so computational accuracy is high on the priority list for this particular component. A master slave design to assure accuracy would be desirable here. The following diagram illustrates this design.

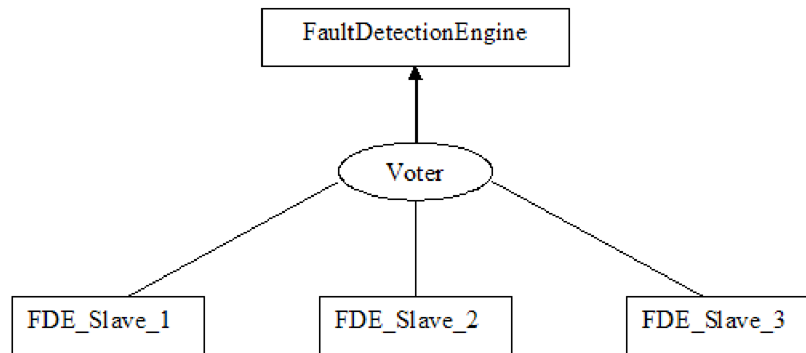

Figure 5.5: Master slave pattern

This design improves calculation accuracy and leads to a more reliable system.

5.1.6 Achieving Fault Tolerance

It is very important to recognize and act on faults appropriately. One way of implementing fault handling is by making the system fault tolerant. Systems can vary from single fault tolerant one out of two systems to large two out of four systems. At the same time there can be systems where only specific components are designed for fault tolerance. In this case we look at a very specific part of the system.

It is the responsibility of the SensorInformation component to acquire both digital and analog information from the sensors. It is assumed that various sensors are used to obtain information from each part of the plant. Each sensor gives back information and has an associated status. Sanity and consistency checks are then performed on the obtained data. SensorInformation would raise a fault even if one of the sensors is not working correctly. However the number of sensors for a power plant may be very high and it may not always be desirable to do that. It is important that SensorInformation should be able to detect faults in the data and function in a predefined way if faults are found.

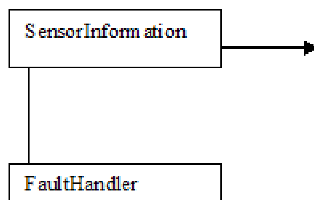

Figure 5.6: Fault Tolerance

A FaultHandler component is added to act as an exception handler. If SensorInformation finds that the data obtained is not consistent then this information is sent to the FaultHandler component. It is then the responsibility of the FaultHandler to mark the faulty or missing sensors as faulty. FaultHandler looks at the list of failed sensors to see if they are safety critical, and if so a fault is raised by SensorInformation. However if not, and if the percent

of failed sensors is below a fixed threshold SensorInformation disregards these marked signals for performing the consistency check. FaultHandler raises an alert about the mal functioning sensors but the system continues to function as usual.

This permits the smooth functioning of the plant since a fault is not raised every time a particular sensor fails.

5.1.7 Unit of Work Pattern

A Unit of Work component maintains a list of objects affected by a transaction and coordinates the writing of changes and resolution of concurreny problems.

For the power plant system the following transformation is done to increase efficiency. The UpdateManager component is responsible for various updates that are performed on the main database. These updates come through from various components like SensorInformation, FaultDiagnosis, FaultDetectionEngine, AlarmManager and AlarmDiagnosis. When altering a database it is very important to keep track of the changes otherwise important data may be lost.

Changes need to be made to the database every time each of the connected components sends through an update. The database can be altered with each change that comes through, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires the database to interact with each component one at a time, while the other com-

ponents wait for their turn, which is impractical if there are multiple requests. An alternative is the creation of an additional UwDb component.

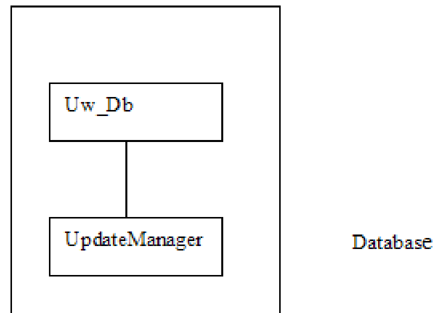

Figure 5.7: Unit of work pattern

The UwDb componet keeps track of all the change requests to the database during a specified period of time. At the end of this time it figures out everything that needs to be done to the database and the appropriate changes are made. So the component basically interacts with various other components of the system to collect change requests and then implements them in groups instead of one at a time. This makes the system more efficient. Additional responsibilities can be added to UwDb. For example, it can go through the list of changes requested and if the same element is changed twice it will only make the most recent change.

5.1.8 Memento pattern

The next transformation is derived from the Memento pattern[7]. This pattern captures and externalizes an objects internal state so the object can be restored to that state later.

It is often necessary to record the internal state of a component. This is required when implementing an undo mechanism that lets users back out of tentative operations or when a system needs to recover from errors. Information about the state of the specific component must be stored elsewhere so it can be restored correctly in case of failure. A memento component stores the snapshot of the internal state of another component. This component is the originator of the memento component. The originator initializes the memento with information that characterizes its current state. The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state. This is also done when the originator fails and needs to go back to a previous state.

For the power plant system the InteractionManager is responsible for interacting with the operator. The operator can send queries to the database through it. The MementoIM componet would hold a snapshot of the state of the InteractionManager. The InteractionManager would act as the originator and initialize MementoIM with its state. Often times an operator may be performing queries in a specific order (say to generate a report) and may choose to undo a mistyped query. This would let the user undo a specific query and return to the previous state.

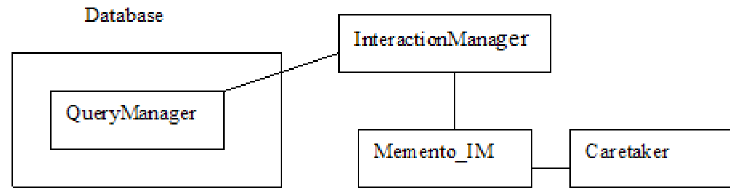

Figure 5.8: Memento pattern

Additionally if the InteractionManager were to fail it could use MementoIM to restore itself to the most previous state. The Caretaker component is responsible for the safekeeping of of MementoIM. This transformation would provide better usability and error handling.

5.1.9 Additional Transformations

The Alarm subcomponent of the architecture has several additional constraints that need to be realized. Some transformations are done to realize these constraints.

Alarm is a highly critical component of the system since it is responsible for drawing attention to faults found. It is imperative that an alarm be raised as soon as a fault is found. Thus the Alarm subcomponent should be very reliable. Therefore there should be two copies of Alarm. Every time one Alarm component fails the other should take the relay. However only one component should work at a time. There is no difference in the importance between the

copies so the switch should occur only in case of failure. AlarmInformation is the only component which requires storing some information so this component should be consistent for the two copies.

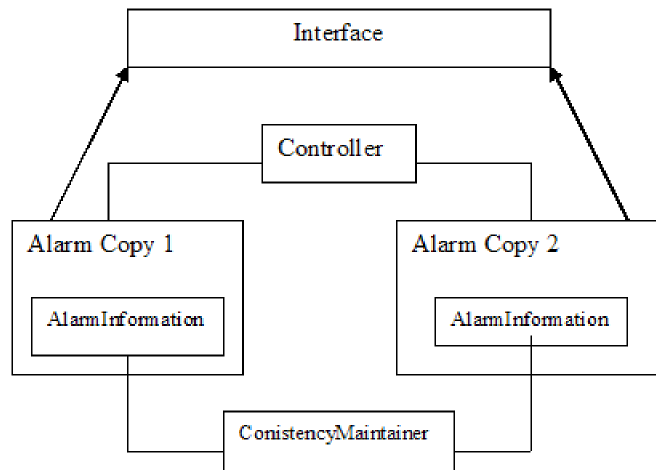

Figure 5.9: Additional transformations pattern

The given figure illustrates how the various constraints are met. There are two copies of Alarm. The controller component ensures that only one is active at a time and switches between the two copies in case of failure. When one of the Alarm component fails the controller also raises the appropriate alert, The active alarm then uses the interface to communicate with the rest of the system. However AlarmInformation needs to be consistent between the two copies. ConsistencyMaintainer assures this. It saves the current AlarmInformation of the active component (say Alarm Copy 1) and in case of failure copies it over to the new active component (Alarm Copy 2). Once the changes

are made (in Alarm Copy 2) it begins to store the current information for the new active component (Alarm Copy 2).

In this case neither of the two copies is more important; control is simply switched to the inactive copy if the active copy fails.

This ensures reliability in the operation and raising of an alarm.

Chapter 6

Conclusion

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

Some of the possible further work in the area includes using data flow and constraints as a basis for logical description on connectors. The connectors were not defined adequately in both architecture approaches, therefore this would be a useful enhancement. We can also look at the kinds of non functional constraints that might apply to data flow elements. We can further explore methods to apply non functional requirements, and study how these requirements affect the architecture. Non functional requirements constitute an important part of a system and this would benefit both designs. For the first method the patterns need to be documented better. It would also be useful to study ways to apply combinations of patterns. For the second method we have

shown how the use of styles and patterns can be provide some non-functional constraints such as reliability and fault tolerance.

Appendices

.1 Goal specifications

The goals are listed following a breadth-first traversal of the goal graph shown in Fig. 1.

- *PerformanceOfThePlantMonitored*

  Def The system must continuously monitor the performance of the plant in order to detect faults in the steam condenser or in the cooling circuit. Moreover, it supports the operators suggesting remedy actions.

  Concerns PowerPlant, SteamCondensor, CoolingCircuit

  RefinedTo FaultDetected, RemedyActionSuggestedWhenFaultDetected, AlarmCorrectlyManaged

- *FaultDetected*

  Def Faults in the steam condenser and in the cooling circuit must be detected

  Concerns SteamCondensor, CoolingCircuit, Fault

  AndRefines PerformanceOfThePlantMonitored

  RefinedTo FaultDetectedInSteamCondensor, FaultDetectedInCooling-Circuit

  FormalDef $\forall f : Fault, \exists! l : Location$

  $Occurs(f, l) \Rightarrow \Diamond Detected(f, l)$

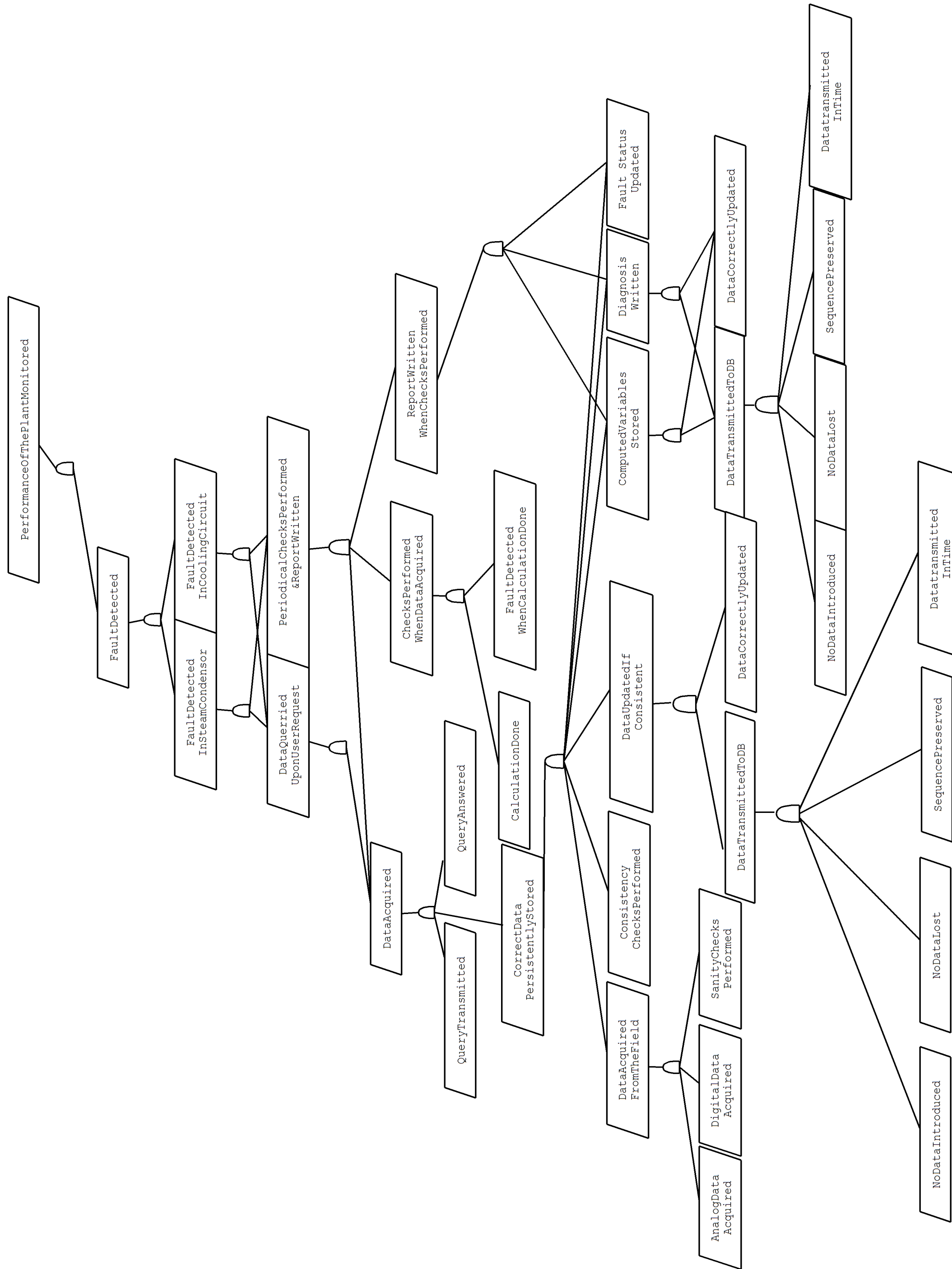

Figure 1: Goal diagram

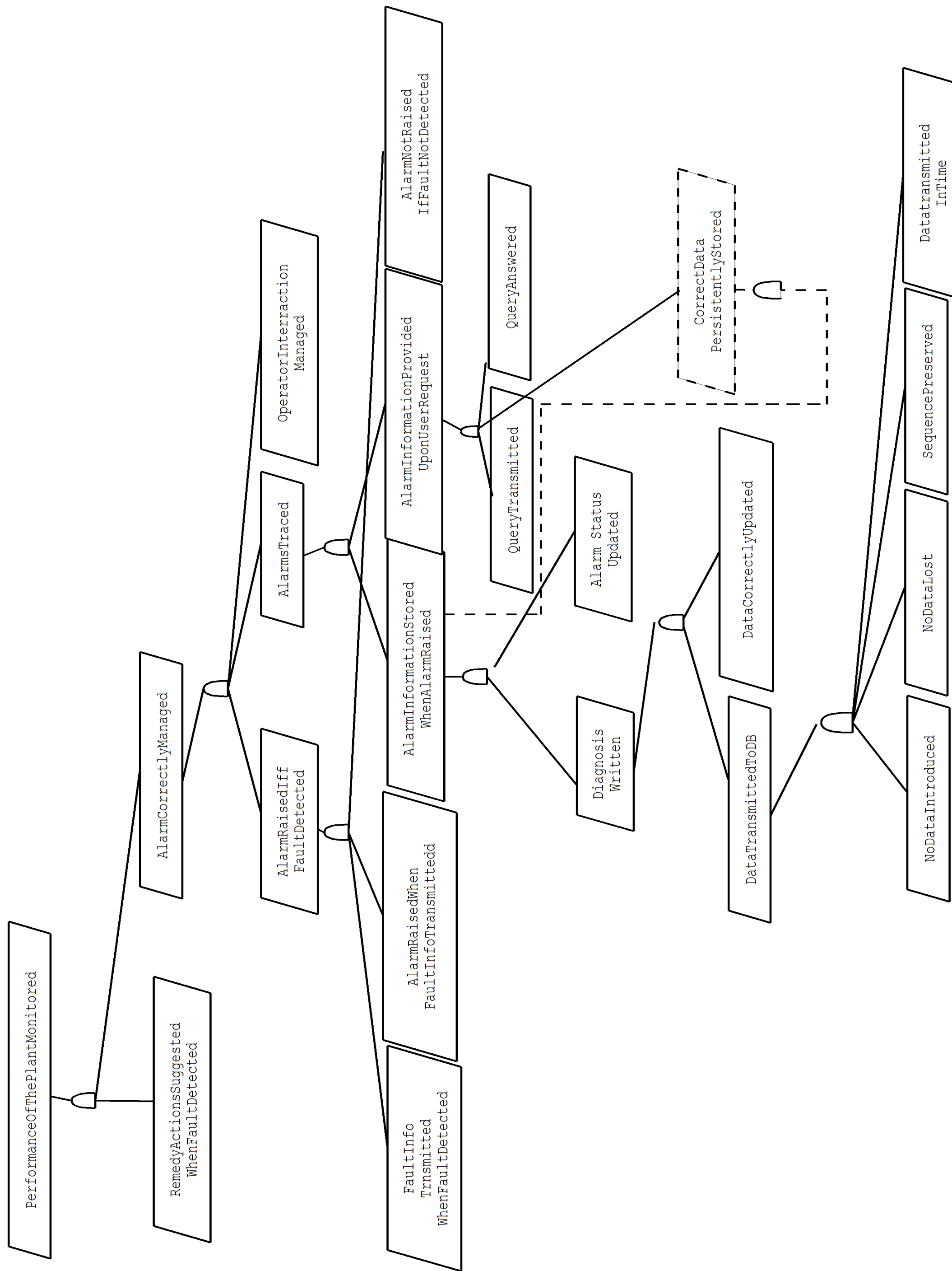

Figure 2: Goal diagram continued

- *RemedyActionsSuggestedWhenFaultDetected*

  Def Remedy actions must be suggested to the operators each time a fault is detected.

  Concerns SteamCondensor, CoolingCircuit, Fault

  AndRefines PerformanceOfThePlantMonitored

  UnderResponsabilityOf PRECON

- *AlarmCorrectlyManaged*

  Def The system must raised an alarm each time a fault is detected. In addition, it must trace and keep the state of all the alarms previously raised.

  Concerns Alarm, Fault

  AndRefines PerformanceOfThePlantMonitored

  RefinedTo AlarmRaisedIffFaultDetected, AlarmTraced, OperatorInterractionManaged

- *FaultDetectedInSteamCondenser*

  Def Faults in the steam condenser must be detected

  Concerns SteamCondenser, Fault

  AndRefines FaultDetected

  RefinedTo DataQuerriedUpondUserRequest, PeriodicalChecksPerformed, ReportsWritten

FormalDef $\forall f : Fault$

$Occurs(f, SteamCondenser) \Rightarrow \Diamond Detected(f, SteamCondenser)$

- *FaultDetectedInCoolingCircuit*

  Def Faults in the cooling circuit must be detected

  Concerns CoolingCircuit, Fault

  AndRefines FaultDetected

  RefinedTo DataQuerriedUpondUserRequest, PeriodicalChecksPerformed, ReportsWritten

  FormalDef $\forall f : Fault$

  $Occurs(f, CoolingCircuit) \Rightarrow \Diamond Detected(f, CoolingCircuit)$

- *AlarmRaisedIffFaultDetected*

  Def The alarm has to be raised if and only if a fault has been detected

  Concerns Alarm

  AndRefines AlarmCorrectlyManaged

  RefinedTo FaultInformationTransmittedWhenFaultDetected, AlarmRaisedWhenFaultInformationTransmitted, AlarmNotRaisedIfFaultNotDetected

  FormalDef $\forall f : Fault, \exists! l : Location, \exists! a : Alarm$

  $Detected(f, l) \Rightarrow \Diamond Raise(f, a)$

  $\wedge \forall a : Alarm, \exists! f : Fault, \exists! l : Location$

  $Raise(f, a) \Rightarrow Detected(f, l)$

- *AlarmTraced*

  Def Informations on alarms previously raised can be retrieved

  Concerns Alarm

  AndRefines AlarmCorrectlyManaged

  RefinedTo AlarmInformationStoredWhenAlarmRaised, AlarmInformationProvidedUponUserRequest

- *DataQuerriedUponUserRequest*

  Def All the data concerning the state of the Power Plant must be provided upon operators request

  Concerns

  AndRefines FaultDetectedInSteamCondensor, FaultDetectedInCoolingCircuit

  RefinedTo CorrectDataPersistentlyStored, QuerryTransmitted, QuerryAnswered

  FormalDef $\forall s : Sensor, \exists! si : SensorInformation$
  $Querry(s) \Rightarrow \Diamond Answer(si) \wedge s \equiv si$

- *PeriodicalChecksPerformed&ReportWritten*

  Def A check must be carried out every 5 minutes in order to detect faults and a report must be written.

  Concerns

AndRefines FaultDetectedInSteamCondensor, FaultDetectedInCoolingCircuit

RefinedTo DataAcquired, ChecksPerformedWhenDataAcquired, ReportWrittenWhenChecksPerformed

FormalDef $\forall f : Fault, \exists! l : Location$

$Occurs(f, l) \Rightarrow \lozenge_{\leq 5min} Detected(f, l)$

- *FaultInformationTransmittedWhenFaultDetected*

Def Each time a Fault is detected, information on that fault has to be transmitted to the ALARM unit

Concerns Alarm

AndRefines AlarmRaisedIffFaultDetected

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall f : Fault, \exists! l : Location, \exists! fi : FaultInformation$

$Detected(f, l) \wedge f \equiv fi \Rightarrow \lozenge Transmitted(fi, PRECON, ALARM)$

- *AlarmRaisedWhenFaultInformationTransmitted*

Def Each time the ALARM unit receive information on a fault, an alarm has to be raised

Concerns Alarm, FaultInformation

AndRefines AlarmRaisedIffFaultDetected

UnderResponsabilityOf ALARM

FormalDef $\forall fi : FaultInformation, \exists! a : Alarm$

$Transmitted(fi, PRECON, ALARM) \Rightarrow \Diamond Raise(fi, a)$

- *AlarmNotRaisedIfFaultNotDetected*

  Def If no fault is detected no alarm can be raised

  Concerns Alarm, Fault

  AndRefines AlarmRaisedIffFaultDetected

  UnderResponsabilityOf ALARM

  FormalDef $\forall a : Alarm, \exists! f : Fault, \exists! l : Location$

  $Raise(f, a) \Rightarrow Detected(f, l)$

- *AlarmInformationStoredWhenAlarmRaised*

  Def Each time an alarm is raised, information on that alarm must be kept in the DataBase.

  Concerns Alarm, AlarmInformation, PowerPlant/AlarmStatus

  AndRefines AlarmTraced

  RefinedTo AlarmDiagnosisWritten, AlarmStatusUpdated

  FormalDef $\forall a : Alarm, \exists! fi : FaultInformation,$

  $\exists! ai : AlarmInformation, \exists! fd : FaultDiagnosis$

  $Raise(fi, a) \wedge a \equiv ai \Rightarrow \Diamond Stored(ai, DB) \wedge Stored(fd, DB) \wedge Concerns(fd, fi, ai) \wedge PowerPlant.AlarmStatus =' on'$

- *AlarmInformationProvidedUponUserRequest*

Def Operators should be able to retrieve informations about all the alarms previously raised

Concerns Alarm, AlarmInformation

AndRefines AlarmTraced

RefinedTo CorrectDataPersistentlyStored, QuerryTransmitted, QuerryAnswered

FormalDef $\forall a : Alarm, \exists! ai : AlarmInformation$

$Querry(a) \Rightarrow \lozenge Answer(ai) \wedge a \equiv ai$

- *DataAcquired*

Def All the data needed are acquired from the field

Concerns Sensor, SensorInformation

AndRefines DataQueriedUponUserRequest, PeriodicalChecksPerformed

RefinedTo CorrectDataPersistentlyStored, QueryTransmitted, QueryAnswered

FormalDef $\forall s : Sensor, \exists! si : SensorInformation$

$Querry(s) \Rightarrow \lozenge_{\leq 2s} Transmitted(si, DB, PRECON) \wedge s \equiv si$

- *ChecksPerformedWhenDataAcquired*

Def Checks must be performed when all the data needed is available in order to detect faults in the Steam Condenser or in the Cooling Circuit

Concerns SensorInformation, Fault

AndRefines PeriodicalChecksPerformed

RefinedTo CalculationDone, FaultDetectedWhenCalculationDone

FormalDef $\forall f : Fault, si : SensorInformation, l : Location$

$Occurs(f, l) \wedge Transmitted(si, DB, PRECON)$

$\Rightarrow \Diamond_{\leq 5min} Detected(f, l)$

$\wedge \neg Occurs(f, l) \wedge Transmitted(si, DB, PRECON)$

$\Rightarrow \Diamond \neg Detected(f, l)$

- *ReportWrittenWhenChecksPerformed*

  Def Whether a fault is detected or not, all the results of the check must be stored.

  Concerns SensorInformation, Fault, FaultInformation

  AndRefines PeriodicalChecksPerformed

  RefinedTo ComputedVariablesStored, DiagnosisWritten, FaultStatusUpdated

  FormalDef $\forall f : Fault, \exists! fi : FaultInformation, \exists! l : Location\ \exists! fd : FaultDiagnosis, \exists! si : SensorInformation\ Detected(f, l)$

  $\Rightarrow \Diamond Stored(fi, DB) \wedge f \equiv fi \wedge Stored(fd, DB) \wedge Concerns(fd, si, fi)$

- *AlarmDiagnosisWritten*

  Def Each time an alarm is raised, information on that alarm must be kept in the DataBase.

Concerns Alarm, AlarmInformation, FaultInformation

AndRefines AlarmInformationStoredWhenAlarmRaised

RefinedTo AlarmDataTransmittedToDB, DataCorrectlyUpdated

FormalDef $\forall a : Alarm, \exists! fi : FaultInformation,$

$\exists! ai : AlarmInformation, \exists! ad : AlarmDiagnosis$

$Raise(fi, a) \Rightarrow \Diamond Stored(ai, DB) \wedge a \equiv ai \wedge Concerns(ad, fi, ai) \wedge$

$Stored(ad, DB)$

- *AlarmStatusUpdated*

  Def If there is at least one alarm raised, the AlarmStatus must be set to on, otherwise it must be set to off.

  Concerns Alarm, Fault, PowerPlant/AlarmStatus

  AndRefines AlarmInformationStoredWhenAlarmRaised

  UnderResponsabilityOf ALARM

  FormalDef $\forall a : Alarm, \exists! fi : FaultInformation$

  $Raise(fi, a) \Rightarrow \circ PowerPlant.AlarmStatus =' on'$

- *DataTransmittedToDB*

  Def Each time an alarm is raised, corresponding information must be transmitted to the DataBase

  Concerns Alarm, AlarmInformation, FaultInformation

  AndRefines AlarmInformationStoredWhenAlarmRaised

RefinedTo NoDataLost, NoDataIntroduce, SequencePreserved, DataTransmittedWithinTimeConstraints

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall a : Alarm, \exists! fi : FaultInformation,$

$\exists! ai : AlarmInformation, \exists! ad : AlarmDiagnosis$

$Raise(fi, a) \wedge a \equiv ai \Rightarrow \Diamond Transmitted(ai, ALARM, DB)$

$\wedge\, Transmitted(ad, ALARM, DB) \wedge Concerns(ad, fi, ai)$

- *DataCorrectlyUpdated*

Def Each time alarm information is transmitted to the DataBase, this information has to be stored

Concerns AlarmInformation, DataBase

AndRefines AlarmInformationStoredWhenAlarmRaised

UnderResponsabilityOf DB

FormalDef $\forall ai : AlarmInformation, ad : AlarmDiagnosis$

$Transmitted(ai, ALARM, DB) \Rightarrow \Diamond Stored(ai, DB)$

$Transmitted(ad, ALARM, DB) \Rightarrow \Diamond Stored(ad, DB)$

- *QueryTransmitted*

Def Each time the operator queries informations on an alarm, the query has to be transmitted to the DataBase

Concerns Alarm, AlarmInformation

AndRefines AlarmInformationProvidedUponUserRequest

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall a : Alarm$

$Querry(a) \Rightarrow Transmitted(a, ALARM, DB)$

- *CorrectDataPersistentlyStored*

Def All the data of the system (reports resulting from checks, alarm information, status of the I/O devices, values of the sensors,etc.) must be stored persistently)

Concerns AlarmInformation, FaultInformation, SensorInformation

AndRefines DataAcquired, AlarmInformationProvidedUponUserRequest

RefinedTo DataAcquiredFromTheField, ConsistencyCheckPerformed, DataUpdatedWhenAcquired, ComputedVariablesStored, DiagnosisWritten, I/OStatusUpdated, AlarmInformationStoredWhenAlarmRaised

FormalDef $\forall si : SensorInformation, fi : FaultInformation, ai : AlarmInformation, fd : FaultDiagnosis, ad : AlarmDiagnosis$

$Stored(si, DB) \wedge Stored(fi, DB) \wedge Stored(ai, DB) \wedge Stored(fd, DB \wedge Stored(ad, DB$

- *CalculationDone*

Def All the calculations needed to detect fault in the PowerPlant are done

Concerns SensorInformation

AndRefines ChecksPerformedWhenDataAcquired

UnderResponsabilityOf PRECON

FormalDef $\forall si : SensorInformation$

$Transmitted(si, DB, PRECON) \Rightarrow \Diamond CalculationDone$

- *FaultDetectedWhenCalculationDone*

  Def When the calculations are done, all the faults present either in the cooling circuit or in the steam condenser must be detected

  Concerns Fault, SteamCondenser, CoolingCircuit

  AndRefines ChecksPerformedWhenDataAcquired

  UnderResponsabilityOf PRECON

  FormalDef $\forall f : Fault, l : Location$

  $CalculationDone \wedge Occurs(f, l) \Rightarrow \Diamond Detected(f, l)$

  $\wedge\ CalculationDone \wedge \neg Occurs(f, l) \Rightarrow \Diamond \neg Detected(f, l)$

- *DataAcquiredFromTheField*

  Def Data concerning the state of the power plant must be acquired

  Concerns Sensor

  AndRefines CorrectDataPersistentlyStored

  RefinedTo AnalogDataAcquired, DigitalDataAcquired, SanityCheck-Performed

FormalDef $\forall s : Sensor$

$s.type =' Digital' \vee s.type =' Analog' \Rightarrow \Diamond Acquired(s)$

- *ConsistencyCheckPerformed*

  Def Consistency checks are performed on all the acquired data in order to ensure consistency within all the sensor datas

  Concerns SensorInformation

  AndRefines CorrectDataPersistentlyStored

  UnderResponsabilityOf ACQUISITION UNIT

  FormalDef $\forall s : Sensor$

  $Acquired(s) \Rightarrow \Diamond Consistent(s)$

- *DataUpdatedWhenAcquired*

  Def When the data have been acquired, they must be stored correctly

  Concerns Sensor, SensorInformation

  AndRefines CorrectDataPersistentlyStored

  UnderResponsabilityOf DB

  FormalDef $\forall si : Sensor$

  $Acquired(s) \wedge Consistent(s) \Rightarrow \Diamond Stored(si, DB) \wedge s \equiv si$

- *ComputedVariablesStored*

  Def

Concerns

AndRefines

RefinedTo

FormalDef

- *FaultDiagnosisWritten*

  Def Each time a fault is detected, informations concerning the fault and the diagnosis must be written

  Concerns SensorInformation, Fault

  AndRefines ReportWrittenWhenChecksPerformed

  RefinedTo DataTransmittedToDB, DataCorrectlyUpdated

  FormalDef $\forall f : Fault, \exists! l : Location, \exists! fi : FaultInformation, \exists si : SensorInformation, \exists! fd : FaultDiagnosis$
  $Detected(f, l) \Rightarrow \Diamond Store(fi, DB) \wedge f \equiv fi \wedge Stored(fd, DB) \wedge Concerns(ds, di, si)$

- *FaultStatusUpdated*

  Def If there is a least one fault detected, the FaultStatus must be set to on, otherwise it must be set to off

  Concerns Fault, PowerPlant/FaultStatus

  AndRefines ReportWrittenWhenChecksPerformed

  UnderResponsabilityOf PRECON

FormalDef $\forall f : Fault, \exists! l : Location$

$Detected(f, l) \Rightarrow \circ PowerPlant.FaultStatus =' on'$

- *DataTransmittedToDB*

Def Each time an fault is detected, corresponding information must be transmitted to the DataBase

Concerns Fault , FaultInformation, SensorInformation

AndRefines FaultDiagnosisWritten, ComputedVariablesStored

RefinedTo NoDataLost, NoDataIntroduce, SequencePreserved, DataTransmittedWithinTimeConstraints

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall f : Fault, \exists l : Location, \exists! fi : FaultInformation, \exists! si : SensorInformation \exists! ad : FaultDiagnosis$

$Detected(f, l) \wedge f \equiv fi \Rightarrow \Diamond Transmitted(fi, PRECON, DB) \wedge Transmitted(fd, ALARM, DB) \wedge Concerns(ad, si, fi)$

- *DataCorrectlyUpdated*

Def Each time fault information is transmitted to the DataBase, this information has to be stored

Concerns FaultInformation, DataBase

AndRefines FaultDiagnosisWritten, ComputedVariablesStored

UnderResponsabilityOf DB

FormalDef $\forall fi : FaultInformation, fd : FaultDiagnosis$

$Transmitted(fi, ALARM, DB) \Rightarrow \Diamond Stored(fi, DB)$

$Transmitted(fd, ALARM, DB) \Rightarrow \Diamond Stored(fd, DB)$

- *AnalogDataAcquired*

  Def All the data coming from working analog sensors are acquired

  Concerns Sensor

  AndRefines DataAcquiredFromTheField

  FormalDef $\forall s : Sensor$

  $s.type =' Analog' \wedge s.status =' on' \Rightarrow \Diamond Acquired(s)$

- *DigitalDataAcquired*

  Def All the data coming from working digital sensors are acquired

  Concerns Sensor

  AndRefines DataAcquiredFromTheField

  FormalDef $\forall s : Sensor$

  $s.type =' Digital' \wedge s.status =' on' \Rightarrow \Diamond Acquired(s)$

- *SanityCheckPerformed*

  Def SanityChecks are performed in order to ensure that all working sensors work correctly

  Concerns Sensor

AndRefines DataAcuiredFromTheField

FormalDef $\forall s : Sensor$

$s.workingProperly = false \wedge \bullet s.status =' on' \Rightarrow \circ s.status =' off'$

$\wedge s.workingProperly = true \wedge \bullet s.status =' off' \Rightarrow \circ s.status =' on'$

- *NoDataLost*

Def No data can be lost during the transmission

Concerns SensorInformation, FaultInformation, AlarmInformation

AndRefines DataTransmittedToDB

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall si : SensorInformation, fi : FaultInformation, ai : AlarmInformation, fd : FaultDiagnosis, ad : AlarmDiagnosis, x : Data$

$x \in si \wedge Transmitted(si, _, _) \Rightarrow x \in Transmitted(si)$

$\wedge\, x \in fi \wedge Transmitted(fi, _, _) \Rightarrow x \in Transmitted(fi)$

$\wedge\, x \in ai \wedge Transmitted(ai, _, _) \Rightarrow x \in Transmitted(ai)$

$\wedge\, x \in fd \wedge Transmitted(fd, _, _) \Rightarrow x \in Transmitted(fd)$

$\wedge\, x \in ad \wedge Transmitted(ad, _, _) \Rightarrow x \in Transmitted(ad)$

- *NoDataIntroduce*

Def No data can be introcue during the transmission

Concerns SensorInformation, FaultInformation, AlarmInformation

AndRefines DataTransmittedToDB

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall si : SensorInformation, fi : FaultInformation, ai : AlarmInformation, fd : FaultDiagnosis, ad : AlarmDiagnosis, x : Data$

$Transmitted(si, _, _) \wedge x \in Transmitted(si) \Rightarrow x \in si$

$\wedge Transmitted(fi, _, _) \wedge x \in Transmitted(fi) \Rightarrow x \in fi$

$\wedge Transmitted(ai, _, _) \wedge x \in Transmitted(ai) \Rightarrow x \in ai$

$\wedge Transmitted(fd, _, _) \wedge x \in Transmitted(fd) \Rightarrow x \in fd$

$\wedge Transmitted(ad, _, _) \wedge x \in Transmitted(ad) \Rightarrow x \in ad$

- *SequencePreserved*

Def The order of the data must be preserved during the transmission

Concerns SensorInformation, FaultInformation, AlarmInformation

AndRefines DataTransmittedToDB

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall si : SensorInformation, fi : FaultInformation, ai : AlarmInformation, fd : FaultDiagnosis, ad : AlarmDiagnosis, x, y : Data, \exists u, v : Data$

$x, y \in si \wedge Transmitted(si, _, _) \wedge Before(x, y, si) \Rightarrow u, v \in Transmitted(si) \wedge Before(u, v, si) \wedge x = u \wedge y = v$

$\wedge\ x, y \in fi \wedge Transmitted(fi, _, _) \wedge Before(x, y, fi) \Rightarrow u, v \in$

$Transmitted(fi) \wedge Before(u,v,fi) \wedge x = u \wedge y = v$

$\wedge\ x,y\ \in\ ai \wedge Transmitted(ai,_,_) \wedge Before(x,y,ai)\ \Rightarrow\ u,v\ \in$

$Transmitted(ai) \wedge Before(u,v,qi) \wedge x = u \wedge y = v$

$\wedge\ x,y\ \in\ fd \wedge Transmitted(fd,_,_) \wedge Before(x,y,fd)\ \Rightarrow\ u,v\ \in$

$Transmitted(fd) \wedge Before(u,v,fd) \wedge x = u \wedge y = v$

$\wedge\ x,y\ \in\ ai \wedge Transmitted(ad,_,_) \wedge Before(x,y,ad)\ \Rightarrow\ u,v\ \in$

$Transmitted(ad) \wedge Before(u,v,ad) \wedge x = u \wedge y = v$

- *DataTransmittedWithinTimeConstraints*

Def All the data that need to be transmittend are effectively transmitted to their destination within 2 s

Concerns SensorInformation, FaultInformation, AlarmInformation

AndRefines DataTransmittedToDB

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall si : SensorInformation, fi : FaultInformation, ai : AlarmInformationm, fd : FaultDiagnosis, ad : AlarmDiagnosis$

$\Diamond_{\leq 2s} Transmitted(si,_,_)$

$\wedge \Diamond_{\leq 2s} Transmitted(fi,_,_)$

$\wedge \Diamond_{\leq 2s} Transmitted(ai,_,_)$

$\wedge \Diamond_{\leq 2s} Transmitted(fd,_,_)$

$\wedge \Diamond_{\leq 2s} Transmitted(ad,_,_)$

.2 Object Specifications

- *PowerPlant*

Def Defines the power plant system. Its components include steam condenser and cooling circuit.

Has PowerPlantID: Integer

Type: Hydrolic, Nuclear, Petrol, Gas, Coal

Power: MegaWatt

Location: Address

FaultStatus: on,off

AlarmStatus: on,off

DomInvar $\forall$ p:PowerPlant

p.faultStatus = on $\Leftrightarrow$ ($\exists$ f:Fault,$\exists$ l:Location)(Occurs(f,l) $\wedge$ PartOf(l,p) $\wedge$ f.Corrected = false

p.alarmStatus = on $\Leftrightarrow$ ($\exists$ a:Alarm,$\exists$ l:Location,$\exists$ f:Fault)(Occurs(f,l) $\wedge$ PartOf(l,p) $\wedge$ Raise(f,a) $\wedge$ f.Activated = true

DomInit FaultStatus = off

AlarmStatus = off

- *SteamCondenser*

Def condenses steam. It accounts for temperature, desired temperature and a range, similarly pressure, a desired pressure and a pressure range.

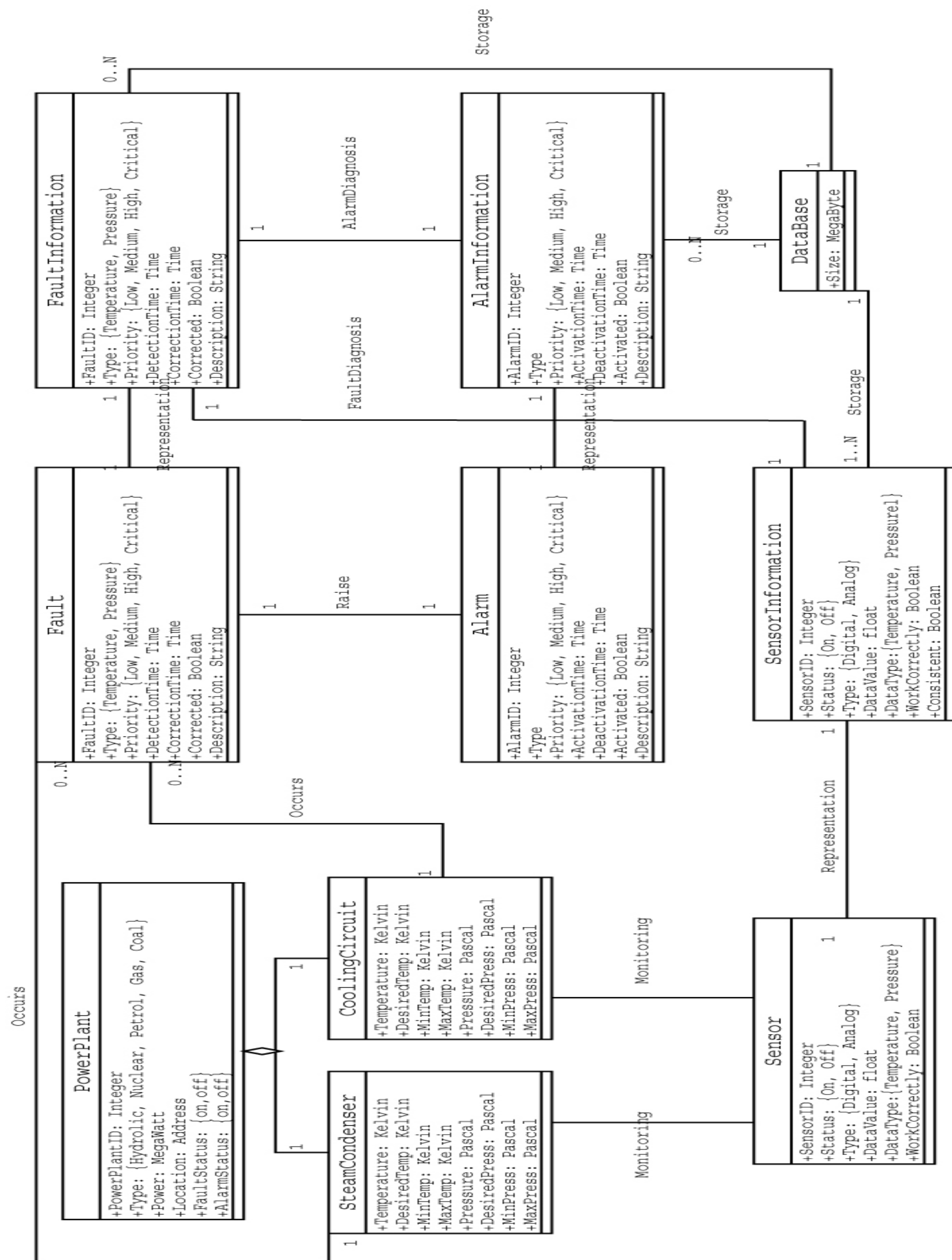

Figure 3: Object diagram

Has Temperature: Kelvin

DesiredTemp: Kelvin

MinTemp: Kelvin

MaxTemp: Kelvin

Pressure: Pascal

DesiredPress: Pascal

MinPress: Pascal

MaxPress: Pascal

DomInvar MinTemp $\leq$ Maxtemp

MinPress $\leq$ MaxPress

DomInit /

- *SteamCondenser*

Def cools the power plant. It is a component of the power plant. It accounts for temperature, desired temperature and a range, similarly pressure, a desired pressure and a pressure range.

Has Temperature: Kelvin

DesiredTemp: Kelvin

MinTemp: Kelvin

MaxTemp: Kelvin

Pressure: Pascal

DesiredPress: Pascal

MinPress: Pascal

MaxPress: Pascal

DomInvar MinTemp $\leq$ Maxtemp

MinPress $\leq$ MaxPress

DomInit /

- *Sensor*

Def it obtains information from the power plant using physically placed sensors. Informations obtained includes data type and its value. Sensors are also checked to ensure that they are working correctly

Has SensorID: Integer

Status: on,off

Type: Digital, Analog

DataValue: Float

DataType: Temperature, Pressure

WorkCorreclty: Boolean

DomInvar $forall$ s: Sensor

s.workingProperly = false $\wedge$ s.status = on $\Rightarrow \circ$ s.status = off

s.workingProperly = true $\wedge$ s.status = off $\Rightarrow \circ$ s.status = on

DomInit status = on

workingProperly = true

- *Fault*

  Def Faults can occur in the cooling circuit or in the steam condenser. When each fault is detected, an ID, type, priority, description and detection time are associated with it. Measures are then taken ot correct the fault.

  Has FaultID: Integer

  Type: Temperature, Pressure

  Priority: Low, Medium, High, Critical

  DetectionTime: Time

  CorrectionTime: Time

  Corrected: Boolean

  Description: String

  DomInvar DetectionTime ¡ CorrectionTime

  Corrected = true $\Rightarrow$ CorrectionTime $\neq$ null

  Corrected = false $\Rightarrow$ CorrectionTime = null

  DomInit DetectionTime = currentTime

  Corrected = false

  CorrectionTime = null

- *Alarm*

  Def An alarm is raised when a fault is detected

  Has AlarmID: Integer

  Type:

Priority: Low, Medium, High, Critical

ActivationTime: Time

DeactivationTime: Time

Activated: Boolean

Description: String

DomInvar ActivationTime ¡ DeactivationTime

Activated = true $\Rightarrow$ DeactivationTime = null

Activated = false $\Rightarrow$ DeactivationTime $\neq$ null

DomInit Activated = true

DeactivationTime = null

- *SensorInformation*

Def representation of the sensor

Has SensorID: Integer

Status: on,off

Type: Digital, Analog

DataValue: Float

DataType: Temperature, Pressure

WorkCorreclty: Boolean

Consistent: Boolean

DomInvar $forall$ s: Sensor

s.workingProperly = false $\wedge$ s.status = on $\Rightarrow \circ$ s.status = off

s.workingProperly = true $\wedge$ s.status = off $\Rightarrow \circ$ s.status = on

DomInit status = on

workingProperly = true

Consistent = true

- *FaultInformation*

Def representation of the fault

Has FaultID: Integer

Type: Temperature, Pressure

Priority: Low, Medium, High, Critical

DetectionTime: Time

CorrectionTime: Time

Corrected: Boolean

Description: String

DomInvar DetectionTime ¡ CorrectionTime

Corrected = true $\Rightarrow$ CorrectionTime $\neq$ null

Corrected = false $\Rightarrow$ CorrectionTime = null

DomInit DetectionTime = currentTime

Corrected = false

CorrectionTime = null

- *AlarmInformation*

  Def representation of the Alarm

  Has AlarmID: Integer

  Type:

  Priority: Low, Medium, High, Critical

  ActivationTime: Time

  DeactivationTime: Time

  Activated: Boolean

  Description: String

  DomInvar ActivationTime ¡ DeactivationTime

  Activated = true ⇒ DeactivationTime = null

  Activated = false ⇒ DeactivationTime ≠ null

  DomInit Activated = true

  DeactivationTime = null

- *DataBase*

  Def A storage unit that hold SensorInformation, AlarmInformation and FaultInformation

  Has Size: Megabytes

  DomInvar /

  DomInit Size = O

.3 Agents Specifications

- `ALARM`

  Def An agent that controls the status of the alarm

  Has AlarmID, Type, Priority, ActivationTime, DeactivationTime, Activated, Description

  Monitors FaultInformation/FaultID, FaultInformation/Type, FaultInformation/Priority, FaultInformation/DetectionTime, FaultInformation/CorrectionTime, FaultInformation/Corrected, FaultInformation/Description

  Controls Alarm/AlarmID, Alarm/Type, Alarm/Priority, Alarm/ActivationTime, Alarm/DeactivationTime, Alarm/Activated, Alarm/Description

  ResponsibleFor AlarmRaisedWhenFaultInfoTransmitted, AlarmNotRaisedIfFaultNotDetected, AlarmStatusUpdated

  DependsOn PRECON

  Perfoms Raise Alarm When Alarm Info Transmitted, Update alarm status, Not Raise Alarm if Fault Not Detected

- `OPERATOR`

  Def Represents user who interacts with the system

  Has /

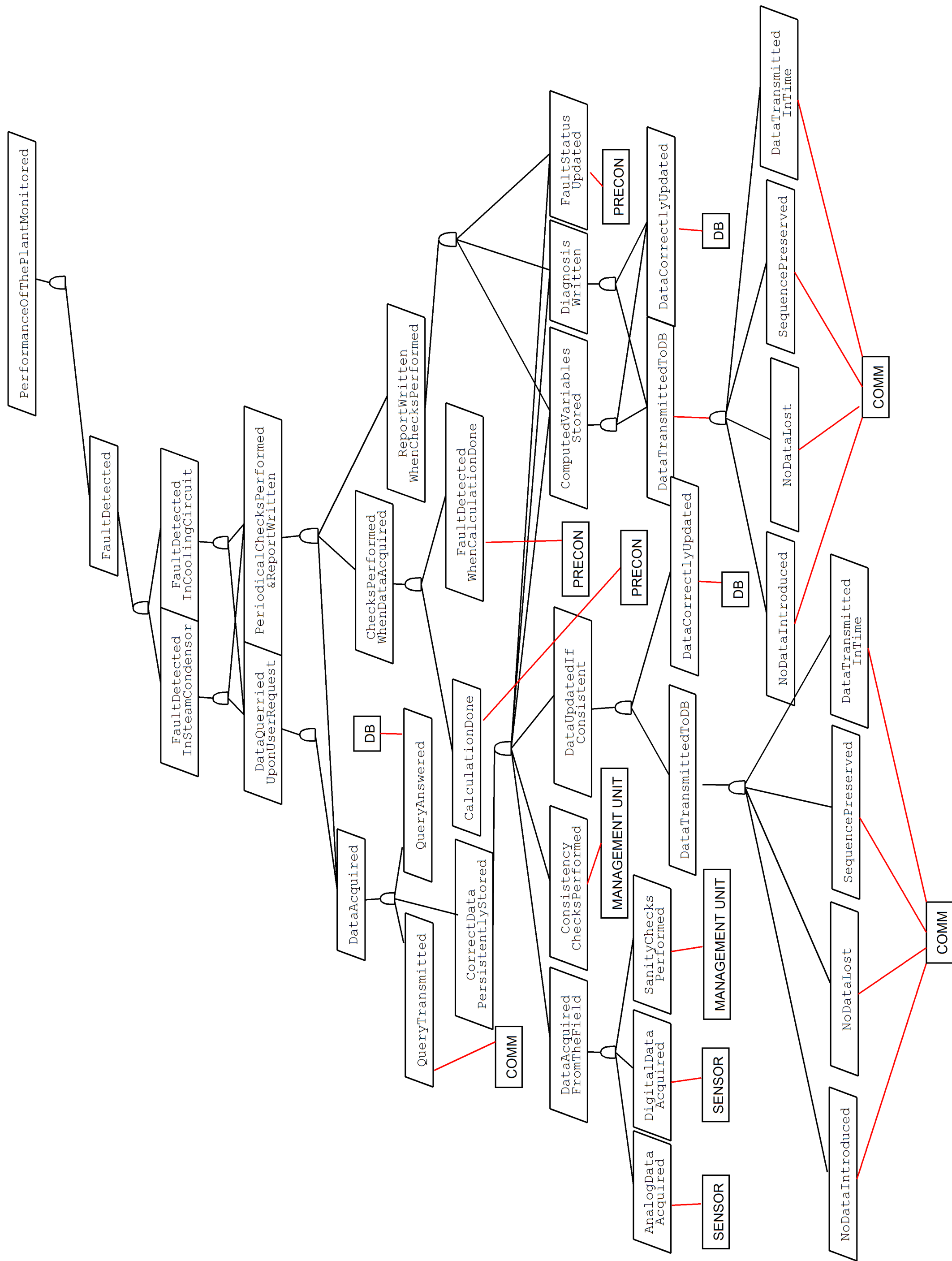

Figure 4: Agent diagram

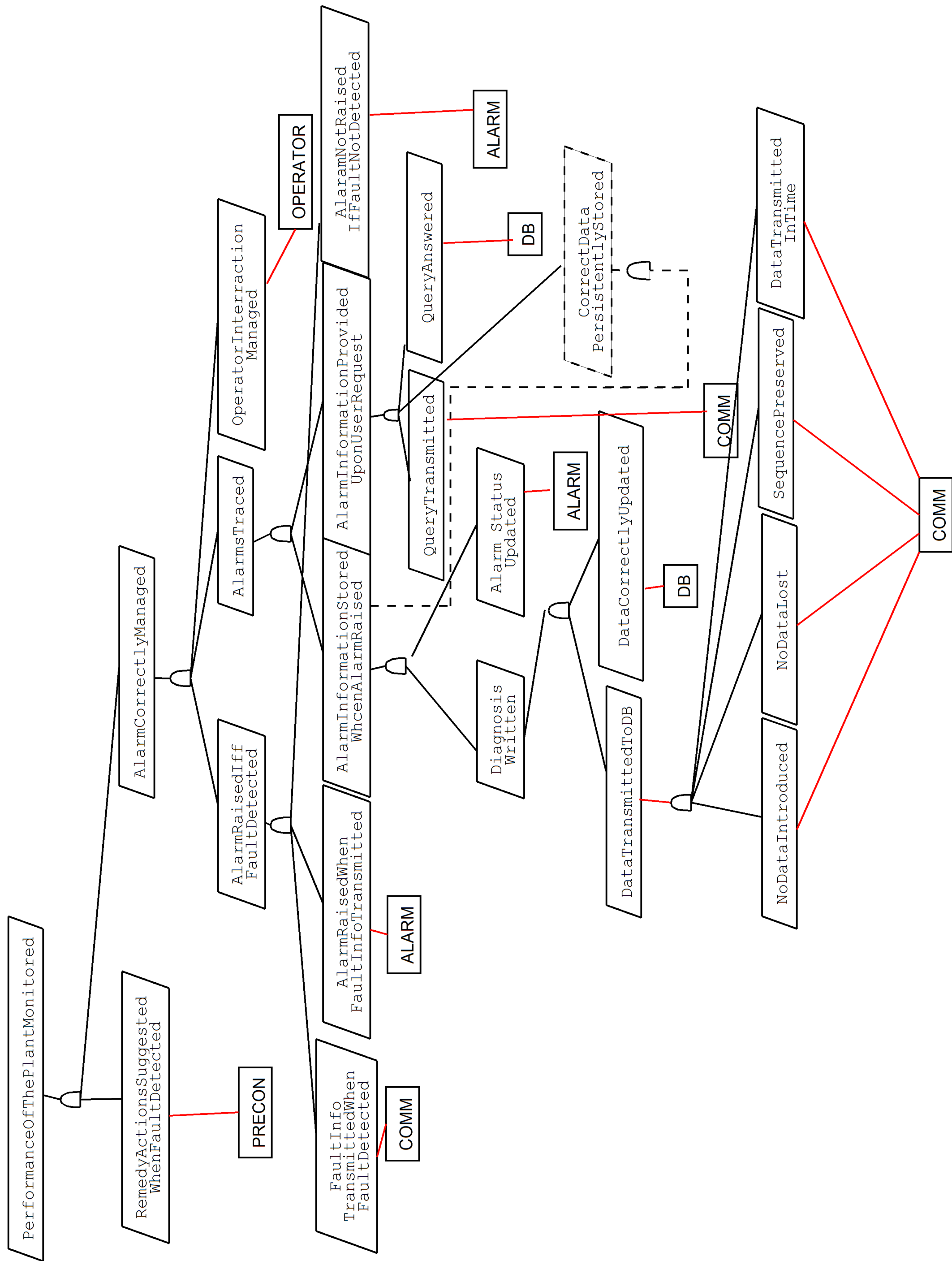

Figure 5: Agent diagram continued

Monitors Alarm/AlarmID, Alarm/Type, Alarm/Priority, Alarm/ActivationTime, Alarm/DeactivationTime, Alarm/Activated, Alarm/Description

Controls /

ResponsibleFor OperatorInteractionsManaged

DependsOn /

Perfoms Manages Operator Interaction

- `DB`

  Def Stores, updates and returns queries on sensor, fault and alarm information

  Has Size

  Monitors FaultInformation/FaultID, FaultInformation/Type, FaultInformation/Priority, FaultInformation/DetectionTime, FaultInformation/CorrectionTime, FaultInformation/Corrected, FaultInformation/Description, AlarmInformation/AlarmID, AlarmInformation/Type, AlarmInformation/Priority, AlarmInformation/ActivationTime, AlarmInformation/DeactivationTime, AlarmInformation/Activated, AlarmInformation/Description, SensorInformation/SensorID, SensorInformation/Status, SensorInformation/Type, SensorInformation/DataValue, SensorInformation/DataType, SensorInformation/WorkProperly

Controls Database/Size

ResponsibleFor DataCorrectlyUpdated, QueryAnswered

DependsOn Communication, PRECON, ALARM, Sensor

Perfoms Update Data Correctly, Answer Query

- `PRECON`

  Def Detects faults from the data and handles fault status

  Has /

  Monitors SensorInformation/SensorID, SensorInformation/Status, SensorInformation/Type, SensorInformation/DataValue, SensorInformation/DataType, SensorInformation/WorkCorrectly, SensorInformation/Consistent

  Controls FaultInformation/FaultID, FaultInformation/Type, FaultInformation/Priority, FaultInformation/DetectionTime, FaultInformation/ CorrectionTime, FaultInformation/Corrected, FaultInformation/Description

  ResponsibleFor CalculationDone, FaultDetectedWhenCalculationDone, RemedyActionSuggestedWhenFaultDetected, FaultStatusUpdated

  DependsOn DataBase

  Perfoms Do Calculation, Detect Fault When Calculation is Done, Suggest Remedy Action When Fault Detected, Update Fault Status

- `COMM`

  Def Handles communication between the different objects

  Has /

  Monitors FaultInformation/FaultID, FaultInformation/Type, FaultInformation/Priority, FaultInformation/DetectionTime, FaultInformation/CorrectionTime, FaultInformation/Corrected, FaultInformation/Description, AlarmInformation/AlarmID, AlarmInformation/Type, AlarmInformation/Priority, AlarmInformation/ActivationTime, AlarmInformation/DeactivationTime, AlarmInformation/Activated, AlarmInformation/Description, SensorInformation/SensorID, SensorInformation/Status, SensorInformation/Type, SensorInformation/DataValue, SensorInformation/DataType, SensorInformation/WorkCorrectly

  Controls /

  ResponsibleFor NoDataIntroduced, NoDataLost, SequencePreserved, DataTransmittedInTime, FaultInfoTransmittedWhenFaultDetected

  DependsOn Sensor, PRECON, ALARM, Database

  Perfoms Transmit Query, Transmit Data to DB, Transmit Fault Info When Fault Detected

- `Sensor`

  Def Physical sensors provide plant information

Has SensorId, Status, Type, DataValue, DataType, WorkCorrectly

Monitors SteamCondensor/Temperature, SteamCondensor/DesiredTemp, SteamCondensor/MinTemp, SteamCondensor/MaxTemp, SteamCondensor/Pressure, SteamCondensor/DesiredPress, SteamCondensor/MinPress, SteamCondensor/MaxPress, CoolingCircuit/Temperature, CoolingCircuit /DesiredTemp, CoolingCircuit /MinTemp, CoolingCircuit /MaxTemp, CoolingCircuit /Pressure, CoolingCircuit /DesiredPress, CoolingCircuit /MinPress, CoolingCircuit /MaxPress, SensorInformation/Status,

Controls Sensor/SensorID, Sensor/Status, Sensor/Type, Sensor/DataValue, Sensor/DataType, SensorInformation/SensorID, SensorInformation/Type, SensorInformation/DataValue, SensorInformation/DataType, SensorInformation/WorkProperly

ResponsibleFor AnalogDataAcquired, DigitalDataAcquired

DependsOn /

Perfoms Acquire Analog Data, Acquire Digital Data

- `MANAGEMENT UNIT`

Def Ensures efficient working of the sensors, checks consistency in data obtained from the sensors

Has /

Monitors SensorInformation/SensorID, SensorInformation/Type, SensorInformation/DataValue, SensorInformation/DataType, SensorInformation/WorkProperly

Controls SensorInformation/Status, SensorInformation/Consistent

ResponsibleFor SanityChecksPerformed, ConsistencyChecksPerformed

DependsOn Sensor

Perfoms Perform Sanity Check, Perform Consistency Check

.4 Operations specifications

- *AcquireAnalogData*

  Def Acquire the data coming from an analog device

  Input s:Sensor,si:SensorInformation

  Output si:SensorInformation/Value

  DomPre s.value $\neq$ si.value

  DomPost s.value = si.value

  ReqTrig for AnalogDataAcquired

  s.value $\neq$ si.value $\mathbf{S}_{=9s}$ s.Type = 'Analog' $\wedge$ s.ID=si.ID $\wedge$ s.Value $\neq$ si.Value

  PerformedBy Sensor

- *AcquireDigitalData*

  Def Acquire the data coming from an digital device

  Input s:Sensor,si:SensorInformation

  Output si:SensorInformation/Value

  DomPre s.value $\neq$ si.value

  DomPost s.value = si.value

  ReqTrig For DigitalDataAcquired

  s.value $\neq$ si.value $\mathbf{S}_{=9s}$ s.Type = 'Digital' $\wedge$ s.ID = si.ID $\wedge$ s.Value $\neq$ si.Value

PerformedBy Sensor

- *SwitchSensorOff*

  Def Turn the sensor off

  Input s:Sensor

  Output s:Sensor/Status

  DomPre s.Status = ’on’

  DomPost s.Status = ’off’

  ReqTrig For SanityCheckPerformed
  ¬ s.WorkingProperly

  PerformedBy ACQUISITION UNIT

- *SwitchSensorOn*

  Def Turn the sensor on

  Input s:Sensor

  Output s:Sensor/Status

  DomPre s.Status = ’off’

  DomPost s.Status = ’on’

  ReqPre For SanityCheckPerformed
  s.WorkingProperly

  Operationalizes SanityCheckPerformed

PerformedBy ACQUISITION UNIT

- *UnValidateData*

  Def Unvalidate the sensor data if they are not considered plausible

  Input si: SensorInformation

  Output si: SensorInformation/Consistent

  DomPre si.Consistent

  DomPost ¬ si.Consistent

  ReqTrig For ConsistencyChecksPerformed

  (si.DataType = 'Temperature' ∧ (si.Value < minTemp ∨ si.Value > maxTemp))
  ∨ (si.DataType = 'Pressure' ∧ (si.Value < minPres ∨ si.Value > maxPres))

  PerformedBy ACQUISITION UNIT

- *ValidateData*

  Def Validate the sensor data if they are considered plausible

  Input si: SensorInformation

  Output si: SensorInformation/Consistent

  DomPre ¬ si.Consistent

  DomPost si.Consistent

ReqPre For ConsistencyChecksPerformed

$\big(\text{si.DataType} = \text{'Temperature'} \wedge (\text{minTemp} \leq \text{si.Value} \leq \text{maxTemp})\big)$

$\vee \big(\text{si.DataType} = \text{'Pressure'} \wedge \text{minPres} \leq \text{si.Value} \leq \text{maxPres})\big)$

PerformedBy ACQUISITION UNIT

- *TransmitSensorData*

Def Transmit the data to the DataBase

Input si: SensorInformation

Output /

DomPre ¬ Transmitted(si,ACQUISITION,DB)

DomPost Transmitted(si,ACQUISITION,DB)

ReqTrig For SensorDataTransmitted

¬ Transmitted(si,ACQUISITION,DB) $\mathbf{S}_{=1s}$ si.Consistent ∧ ¬ Transmitted(si,ACQUISITION,DB)

PerformedBy COMMUNICATION

- *UpdateSensorData*

Def Update the data in the DataBase

Input si: SensorInformation

Output /

DomPre ¬ Stored(si)

DomPost Stored(si)

ReqTrig For SensorDataUpdated

$\neg$ Stored(si) $\mathbf{S}_{=1s}$ Transmitted(si,ACQUISITION,DB)$\wedge \neg$ Stored(si)

PerformedBy DB

- *TransmitSensorQuery*

Def transmit a sensor query to the DataBase

Input s: Sensor

Output /

DomPre $\neg$ Transmitted(s,PRECON,DB)

DomPost Transmitted(s,PRECON,DB)

ReqTrig For SensorQuerryTransmitted

$\neg$ Transmitted(s,PRECON,DB) $\mathbf{S}_{=1s}$ Query(s)

$\wedge \neg$ Transmitted(s,PRECON,DB)

PerformedBy COMMUNICATION

- *AnswerSensorQuery*

Def Answer to a sensor query

Input s: Sensor

Output si: SensorInformation

DomPre $\neg$ Transmitted(si,DB,PRECON)

DomPost Transmitted(si,DB,PRECON)

ReqTrig For SensorQueryAnswered

$\neg$ Transmitted(si,DB,PRECON) $\mathbf{S}_{=1s}$ Transmitted(s,PRECON,DB)$\wedge$ Query(s)$\wedge$ Stored(si) $\wedge$ si.ID = s.ID $\wedge$ $\neg$ Transmitted(si,DB,PRECON)

PerformedBy DB

- *Calculate*

Def calculate all needed things in order to detect faults

Input si: SensorInformation

Output /

DomPre $\neg$ CalculationDone

DomPost CalculationDone

ReqTrig For CalculationDone

$\neg$ CalculationDone $\mathbf{S}_{=1s}$ Transmitted(si,DB,PRECON) $\wedge$ $\neg$ CalculationDone

PerformedBy PRECON

- *DetectFault*

Def detect Fault

Input f: Fault, l: Location

Output /

DomPre $\neg$ Detected(f,l)

DomPost Detected(f,l)

ReqTrig For FaultDetectedWhenCalculationDone

$\neg$ Detected(f,l) $\mathbf{S}_{=1s}$ CalculationDone $\wedge$ Occurs(f,l) $\wedge$ $\neg$ Detected(f,l)

PerformedBy PRECON

- *TransmitDiagnosisData*

Def Transmit the data concerning the diagnosis of a fault to the DataBase

Input f: Fault, l: Location, fi: FaultInformation, si: SensorInformation, fd: FaultDiagnosis

Output /

DomPre $\neg$ Transmitted(fi,PRECON,DB) $\vee$ $\neg$ Transmitted(ad,PRECON,DB) $\vee$ $\neg$ Concerns(ad,si,fi)

DomPost Transmitted(fi,PRECON,DB) $\wedge$ Transmitted(ad,PRECON,DB) $\wedge$ Concerns(ad,si,fi)

ReqTrig For DiagnosisDataTransmitted

$\neg$ Transmitted(fi,PRECON,DB) $\vee$ $\neg$ Transmitted(ad,PRECON,DB) $\vee$ $\neg$ Concerns(ad,si,fi) $\mathbf{S}_{=1s}$ Detected(f,l) $\wedge$ f.ID = fi.ID $\wedge$ $\Big($ $\neg$ Transmitted(fi,PRECON,DB) $\vee$ $\neg$ Transmitted(ad,PRECON,DB) $\vee$ $\neg$ Concerns(ad,si,fi) $\Big)$

PerformedBy COMMUNICATION

- *UpdateDiagnosisData*

Def Store the data concerning a detected fault in the DataBase

Input fi: SensorInformation, fd: FaultDiagnosis

Output /

DomPre $\neg$ Stored(fi) $\vee$ $\neg$ Stored(fd)

DomPost Stored(fi) $\wedge$ Stored(fd)

ReqTrig For DiagnosisDataUpdated

$\neg$ Stored(fi) $\vee$ $\neg$ Stored(fd) $\mathbf{S}_{=1s}$ Transmitted(fd,PRECON,DB) $\wedge$

Transmitted(fi,PRECON,DB) $\wedge$ $\big(\neg$ Stored(fi) $\vee$ $\neg$ Stored(fd)$\big)$

PerformedBy DB

- *SwitchFaultStatusOn*

Def switch the Fault Status on

Input f: Fault, l: Location, PowerPlant

Output PowerPlant/FaultStatus

DomPre PowerPlant.FaultStatus = off

DomPost PowerPlant.FaultStatus =on$\neg$

Transmitted(fi,PRECON, ALARM)

ReqTrig For FaultStatusUpdated

Detected(f,l)

PerformedBy PRECON

- *SwitchFaultStatusOff*

Def switch the Fault Status off

Input f: Fault, l: Location, PowerPlant

Output PowerPlant/FaultStatus

DomPre PowerPlant.FaultStatus = on

DomPost PowerPlant.FaultStatus = off

ReqPre For FaultStatusUpdated

¬ Detected(f,l)

PerformedBy PRECON

- *TransmitFaultInformation*

Def Transmit Fault Information to The ALARM Management unit

Input f: Fault, l: Location, fi: FaultInformation

Output /

DomPre ¬ Transmitted(fi,PRECON, ALARM)

DomPost Transmitted(fi,PRECON, ALARM)

ReqTrig For FaultInformationTransmittedWhenFaultDetected

¬ Transmitted(fi,PRECON, ALARM) $\mathbf{S}_{=1s}$ Detected(f,l) ∧ f.ID = fi.ID ∧ ¬ Transmitted(fi,PRECON, ALARM)

PerformedBy COMMUNICATION

- *RaiseAlarm*

Def Raise the alarm

Input fi: FaultInformation

Output a: Alarm

DomPre $\neg$ Raise(fi,a)

DomPost Raise(fi,a)

ReqTrig For AlarmRaisedWhenFaultInformationTransmitted

$\neg$ Raise(fi,a) $\mathbf{S}_{=1s}$ Transmitted(fi,PRECON, ALARM) $\wedge$ $\neg$ Raise(fi,a)

PerformedBy ALARM

- *TransmitAlarmData*

Def Transmit the alarm data to the DataBase

Input fi: FaultInformation, a: Alarm, ai: AlarmInformation, ad: AlarmDiagnososis

Output /

DomPre $\neg$ Transmitted(ai,ALARM,DB) $\vee$ $\neg$ Transmitted(ad,ALARM,DB) $\vee$ $\neg$ Concerns(ad,fi,ai)

DomPost Transmitted(ai,ALARM,DB) $\wedge$ Transmitted(ad,ALARM,DB) $\wedge$ Concerns(ad,fi,ai)

ReqTrig For AlarmDataTransmitted

$\neg$ Transmitted(ai,ALARM,DB) $\vee$ $\neg$ Transmitted(ad,ALARM,DB) $\vee$ $\neg$ Concerns(ad,fi,ai) $\mathbf{S}_{=1s}$ Raise(fi,a) $\wedge$ a.ID = ai.ID $\wedge$ $\Big($ $\neg$ Transmitted(ai,ALARM,DB) $\vee$ $\neg$ Transmitted(ad,ALARM,DB) $\vee$ $\neg$ Concerns(ad,fi,ai) $\Big)$

PerformedBy COMMUNICATION

- *UpdateAlarmData*

  Def Update Alarm data in the DataBase

  Input ai: AlarmInformation, ad: AlarmDiagnosis

  Output /

  DomPre ¬ Stored(ai) ∨ ¬ Stored(ad)

  DomPost Store(ai) ∧ Stored(ad)

  ReqTrig For AlarmDataCorrectlyUpdated

  ¬ Stored(ai) ∨ ¬ Stored(ad) $\mathbf{S}_{=1s}$ Transmitted(ai,ALARM,DB) ∧ Transmitted(ad,ALARM,DB) ∧ (¬ Stored(ai) ∨ ¬ Stored(ad))

  PerformedBy

- *SwitchAlarmStatusOn*

  Def switch the Alarm Status on

  Input a: Alarm, fi: FaultInformation, PowerPlant

  Output PowerPlant/AlarmStatus

  DomPre PowerPlant.AlarmStatus = off

  DomPost PowerPlant.AlarmStatus = on

  ReqTrig For AlarmStatusUpdated

  Raise(fi,a)

Operationalizes AlarmStatusUpdated

PerformedBy ALARM

- *SwitchAlarmStatusOff*

  Def switch the Alarm Status off

  Input a: Alarm, fi: FaultInformation, PowerPlant

  Output PowerPlant/AlarmStatus

  DomPre PowerPlant.AlarmStatus = on

  DomPost PowerPlant.AlarmStatus = off

  ReqPre For AlarmStatusUpdated
  ¬ Raise(fi,a)

  Operationalizes AlarmStatusUpdated

  PerformedBy ALARM

- *TransmitAlarmQuery*

  Def transmit a alarm query to the DataBase

  Input a: Alarm

  Output /

  DomPre ¬ Transmitted(a,ALARM,DB)

  DomPost Transmitted(a,ALARM,DB)

ReqTrig For AlarmQuerryTransmitted

$\neg$ Transmitted(a,ALARM,DB) $\mathbf{S}_{=1s}$ Query(a) $\wedge$

$\neg$ Transmitted(a,ALARM,DB)

PerformedBy COMMUNICATION

- *AnswerAlarmQuery*

Def Answer to a alarm query

Input a: Alarm

Output ai: AlarmInformation

DomPre $\neg$ Transmitted(ai,DB,ALARM)

DomPost Transmitted(ai,DB,ALARM)

ReqTrig For AlarmQueryAnswered

$\neg$ Transmitted(ai,DB,ALARM) $\mathbf{S}_{=1s}$ Transmitted(a,ALARM,DB)$\wedge$

Query(a)$\wedge$ Stored(ai) $\wedge$ ai.ID = a.ID $\wedge$ $\neg$ Transmitted(ai,DB,ALARM)

PerformedBy DB

.5 Axel van Lamsweerde Architecture

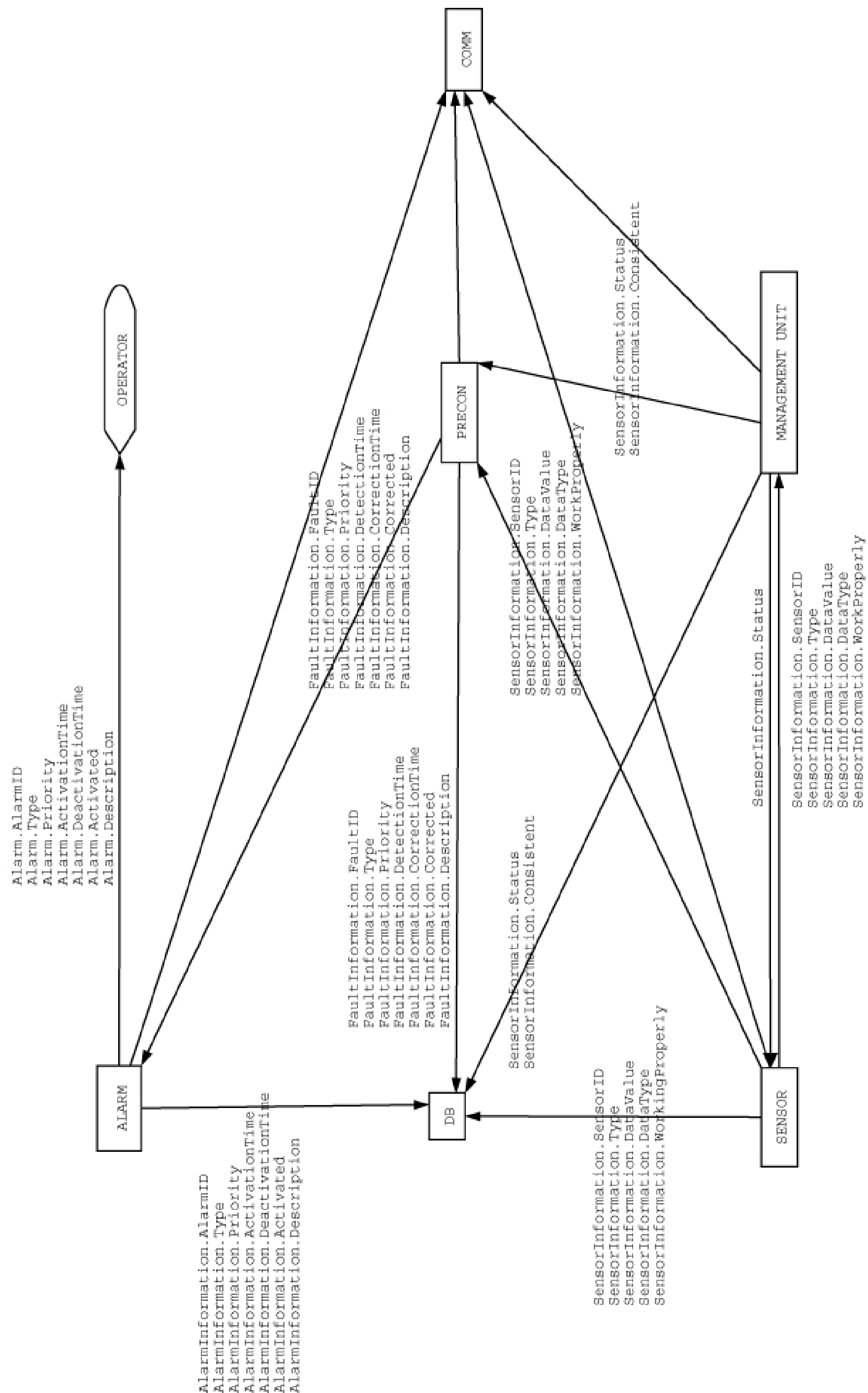

Figure 6: Step 1: dataflow architecture

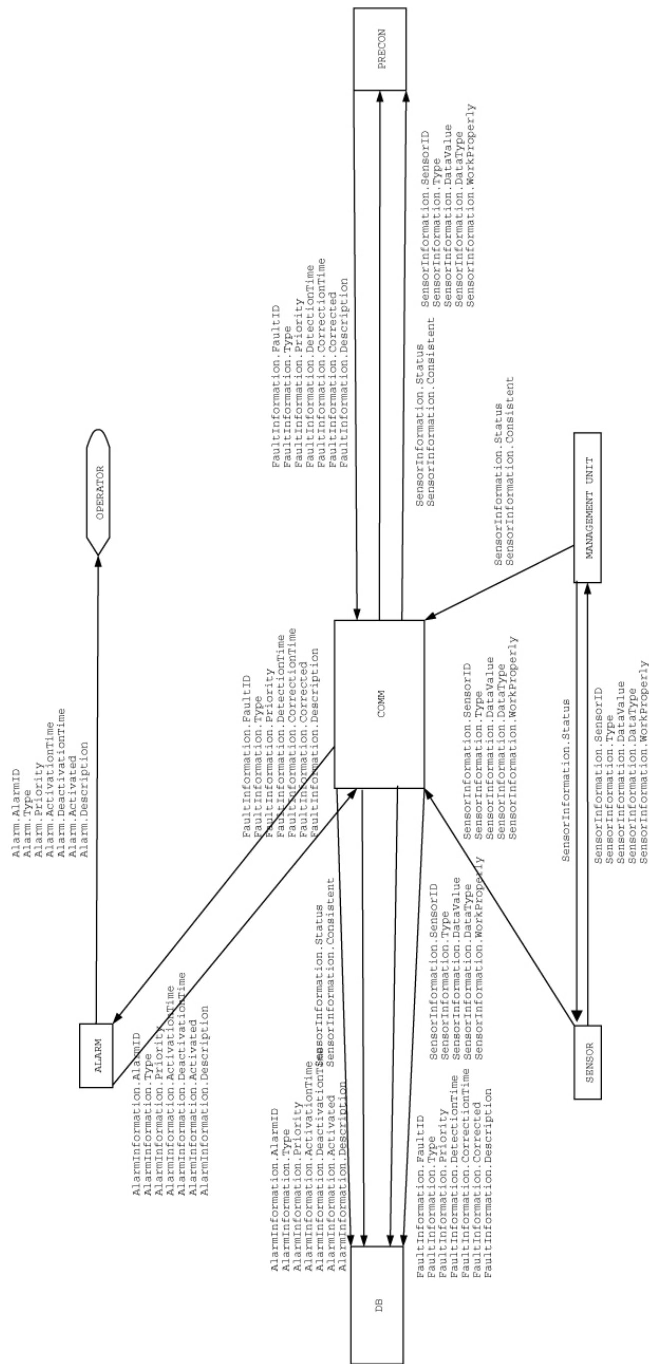

Figure 7: Step 2: style-based refined architecture

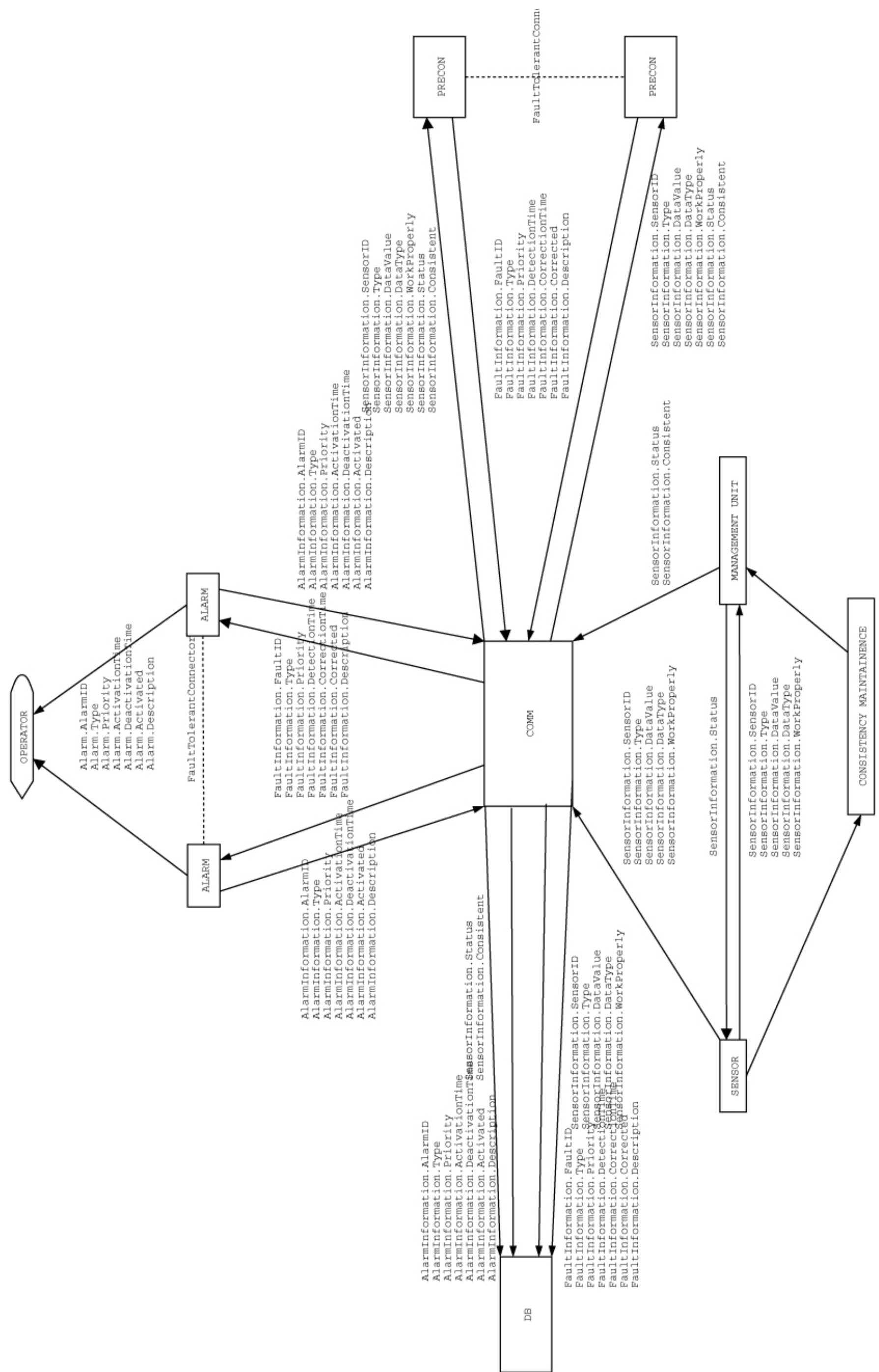

Figure 8: Step 3: pattern-based refined architecture

.6 Architecture Prescriptions

Preskriptor Specification: PowerPlant Monitoring System

Problem Goals Specifications: PowerPlant Monitoring Process

Components:

- **Component** PowerPlantMonitoringSystem

  Type Processing

  Constraints PerformancOfThePlantMonitored

  Composed of PRECON

  ALARM

  DataBase

  Communication

  Uses /

- **Component** PRECON

  Type Processing

  Constraints FaultDetected

  RemedyActionSuggested

  PeriodicalChecksPerformed&ReportWritten

  Composed of FaultDetectionEngine

  FaultInformation

  FaultDiagnosis

  SensorInformation

  SensorConnect

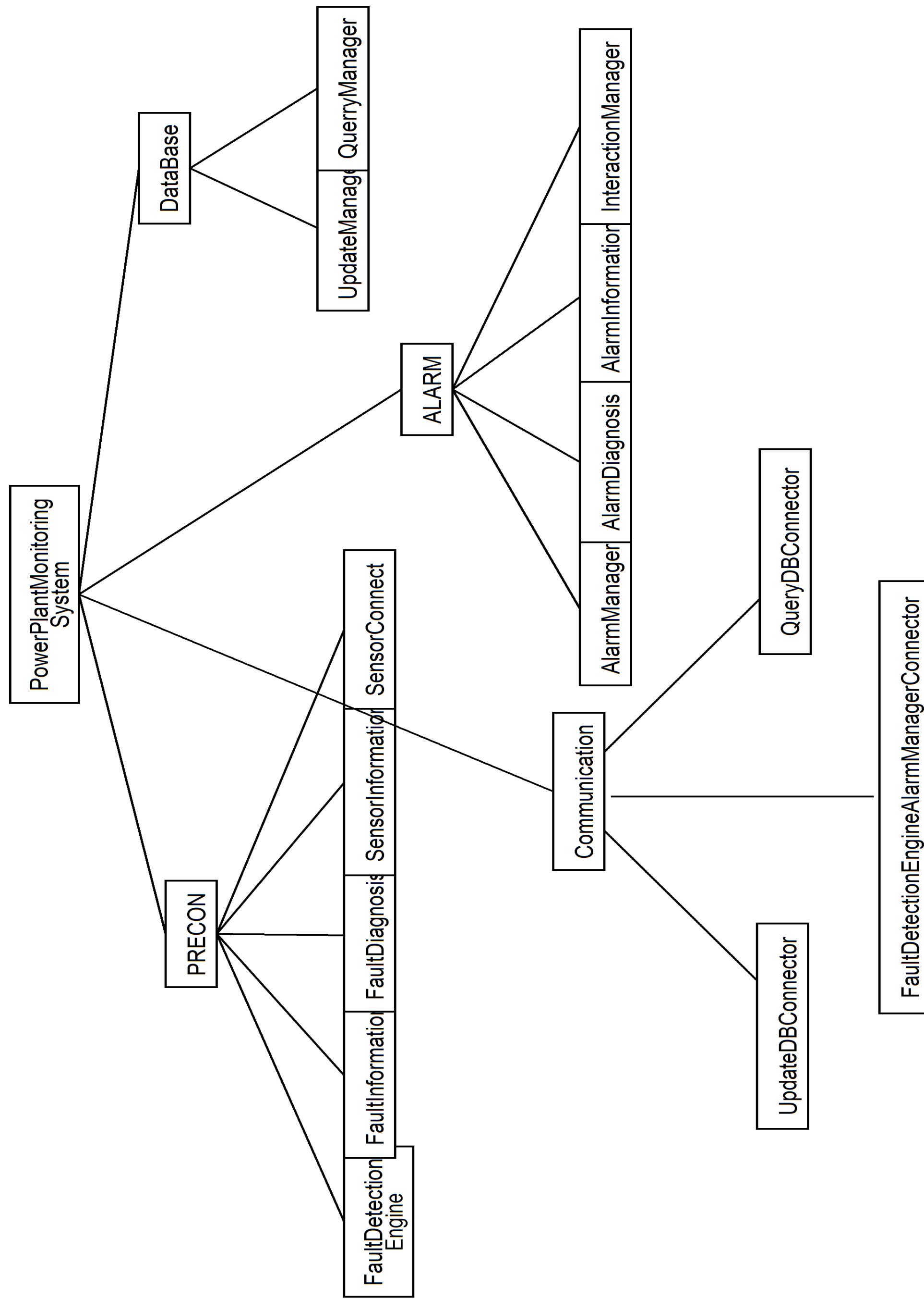

Figure 9: Component refinment tree

Uses /

- **Component** ALARM

  Type Processing

  Constraints AlarmCorrectlyManaged

  AlarmRaisedIffFaultDetected

  AlarmTraced

  Composed of AlarmManager

  AlarmInformation

  AlarmDiagnosis

  InteractionManager

  Uses /

- **Component** Database

  Type Processing

  Constraints CorrectDataPersistentlyStored

  Composed of QueryManager

  UpdateManager

  Uses /

- **Component** Communication

  Type Connector

  Constraints NoDataIntroduced

  NoDataLost

  SequencePreserved

DataTransmittedInTime

DataTransmittedToTheDB

QueryTransmitted

FaultInformationTransmittedWhenFaultDetected

Composed of UpdateDBConnect

QueryDBConnect

FaultDetectionEngineAlarmManagerConnect

Uses /

- **Component** FaultDetectionEngine

  Type Processing

  Constraints CalculationDone

  FaultDetectedWhenCalculationDone

  FaultStatusUpdated

  CheckPerformedWhenDataAcquired

  ReportWrittenWhenCheckPerformed

  Composed of /

  Uses SensorConnect *to interract with* SensorInformation

  FaultDetectionEngineAlarmManagerConnect *to interract with* AlarmManager

  UpdateDBConnect *to interract with* UpdateManager

- **Component** FaultInformation

  Type Data

Constraints FaultInformationTransmittedWhenFaultDetected

Composed of /

Uses FaultDetectionEngineAlarmManagerConnect

to interract with AlarmManager

UpdateDBConnect *to interract with* UpdateManager

- **Component** FaultDiagnosis

  Type Data

  Constraints DiagnosisWritten

  ComputedVariablesStored

  Composed of /

  Uses UpdateDBConnect *to interract with* DBUpdateManager

- **Component** SensorInformation

  Type Data

  Constraints AnalogDataAcquired

  DigitalDataAcquired

  SanityCheckPerformed

  ConsistencyCheck

  Composed of /

  Uses SensorConnect *to interract with* DB

  SensorConnect *to interract with* FaultDetectionEngine

- **Component** SensorConnect

  Type Connector

Constraints DataAcquiredFromTheField

Composed of /

Uses /

- **Component** UpdateDBConnect

  Type Connector

  Constraints Secure

  TimeConstraint = 2s

  Composed of /

  Uses /

- **Component** QueryDBConnect

  Type Connector

  Constraints TimeConstraint = 5s

  Composed of /

  Uses /

- **Component** FaultDetectionEngineAlarmManagerConnect

  Type Connector

  Constraints FaultTolerant

  Secure

  TimeConstraint = 1s

  Composed of /

  Uses /

- **Component** AlarmManager

  Type Processing

  Constraints AlarmRaisedWhenFaultInformationTransmitted
  FaultInformationTransmitted
  AlarmStatusUpdated
  AlarmNotRaisedIfFaultNotDetected

  Composed of /

  Uses FaultDetectionEngineAlarmManagerConnect *to interract with* FaultDetectionEngine UpdateDBConnect *to interract with* UpdateManager

- **Component** AlarmInformation

  Type Data

  Constraints AlarmInformationStoredWhenAlarmRaised

  Composed of /

  Uses UpdateDBConnect *to interract with* UpdateManager

- **Component** AlarmDiagnosis

  Type Data

  Constraints DiagnosisWritten

  Composed of /

  Uses UpdateDBConnect *to interract with* UpdateManager

- **Component** InteractionManager

  Type Processing

Constraints OperatorInteractionManaged

Composed of /

Uses QueryDBConnect *to interract with* QueryManager

- **Component** QueryManager

  Type Processing

  Constraints QueryAnswered

  DataQueriedUponUserRequest

  AlarmInformationProvidedUponUserRequest

  DataAcquired

  Composed of /

  Uses QueryDBConnect *to interract with* InteractionManager

- **Component** UpdateManager

  Type Processing

  Constraints DataCorrectlyUpdated DataUpdatedIfConsistent

  Composed of /

  Uses SensorConnect *to interact with* SensorInformation

  UpdateDBConnect *to interact with* FaultDetectionEngine

  UpdateDBConnect *to interact with* FaultDiagnosis

  UpdateDBConnect *to interact with* AlarmManager

  UpdateDBConnect *to interact with* AlarmDiagnosis

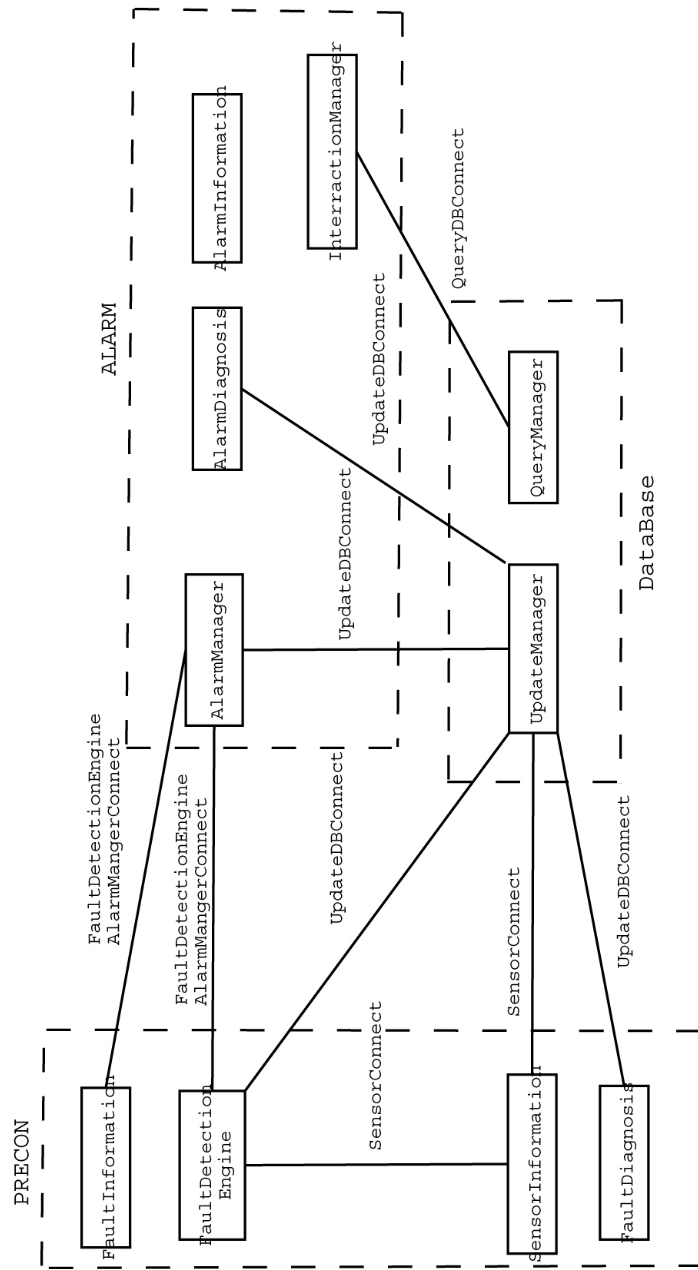

Figure 10: Box diagram of the architecture

.7 Additional constraints on the system

.7.1 Constraints on the Database

1. **Informal Def :** Every update on the main database has to be done on the backup database

   Formal Def : $\forall$ x:Data Update(x,mainDB) $\Rightarrow \Diamond$ Update(x,backupDB)

2. **Informal Def :** No additional update should to be made

   Formal Def : Update(x,backupDB) $\wedge$ mainDB.Status = working $\Rightarrow$ Update(x,mainDB)

3. **Informal Def :** If the main database fails the backup database should take the relay

   Formal Def : mainDB.Status = failure $\wedge$ backupDB.Status=working $\Rightarrow \circ \neg$ mainDB.work $\wedge$ backupDB.work

4. **Informal Def :** If the main database recovers after a failure all the updates made on the backup database have to be done on the main database. The main database has also to reused instead of the backup one.

   Formal Def : $\forall$ x:Data Update(x,backupDB) $\wedge \bullet$ mainDB.Status = failure $\wedge$ mainDB.Status = working $\Rightarrow$ Update(x,mainDB)

5. **Informal Def :** No Query on something that is currently updated can be performed

Formal Def : $\forall$ x:Data Query(x) $\Rightarrow$ $\big($ $\neg$ Update(x,mainDB) $\wedge$ mainDB.Work $\big)$ $\vee$ $\big($ $\neg$ Update(x,backupDB) $\wedge$ backupDB.Work $\big)$

6. **Informal Def :** Only one database can work at a time

   Formal Def : mainDB.Work $\Rightarrow$ $\neg$ backupDB.Work $\wedge$ backupDB.Work $\Rightarrow$ $\neg$ mainDB.Work

.7.2 Constraints on the connector between ALARM & PRECON (i.e., FaultDetectionEngineAlarmManagerConnect)

1. **Informal Def :** There has to be two copies of PRECON and ALARM

   Formal Def : $\forall$ x: Component x.type = PRECON $\vee$ x.type = ALARM $\Rightarrow$ $\exists$ y:Component x.type = y.type $\wedge$ $\neg$ x = y $\wedge$ x $\equiv$ y

2. **Informal Def :** Every time a component fails (PRECON or ALARM), the copy should take te relay

   Formal Def : $\forall$ x:Component $\big($x.type = PRECON $\vee$ x.type = ALARM $\big)$ $\wedge$ x.Status = failure $\Rightarrow$ $\exists$ y:Component x.type = y.type $\wedge$ y.Status = working $\wedge$ $\circ$ $\big($ y.Work $\wedge$ $\neg$ x.Work $\big)$

3. **Informal Def :** Only one component (PRECON or ALARM) should be working at a time

   Formal Def : $\forall$ x:Component $\big($ x.type = PRECON $\vee$ x.type = ALARM $\big)$ $\wedge$ x.Work $\Rightarrow$ $\neg$ $\exists$ y:Component x.type=y.type $\wedge$ $\neg$ x = y $\wedge$ y.Work

4. **Informal Def :** There is no difference in importance between the copies. So the switch should only occur in case of a failure

   Formal Def : $\bullet$ $\neg$ x.Work $\wedge$ x.Work $\Rightarrow$ $\exists$ y $\bullet$ y.status=working $\wedge$ y.status=failure $\wedge$ x.type=y.type $\wedge$ $\neg$ x = y $\wedge$ x $\equiv$ y

5. **Informal Def :** A failure of PRECON or ALARM should not affect the other. The other should continue to work fine

   Formal Def : $\exists$ x:Component $\bullet$ x.Status = working $\wedge$ x.Status=failure $\Rightarrow$ $\big($ $\forall$ y:Component x.type $\neq$ y.type $\wedge$ $\bullet$ y.Satus=working $\Rightarrow$ y.Status =woking $\big)$

.8 Goal Oriented Requirements to Architecture Prescription - Updated

The Brandozzi Perry method converts the goal oriented requirement specifications of KAOS into architectural prescriptions.

The components in an architecture prescription can be of three different types - process, data or connector. Processing components perform transformation the data components. The data components contain the necessary information. The connector components, which can be implemented by data or processing components, hold the system together. All components are characterized by goals that they are responsible for. The interactions and restrictions of these components characterize the system.

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

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.

The First Step

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 defined as *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 preventing a specification that constrains the lower level designs.

In this case 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 sub systems 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.

An example of this can be seen in the Paper selection process[1] shown below.

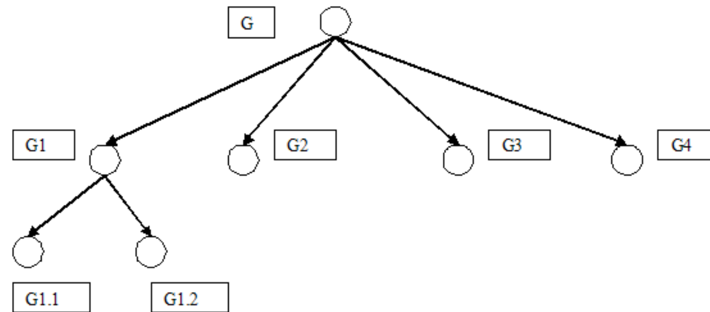

Figure 11: Goal refinement tree for the paper selection process

G:Maintain[QualityOfTheScientificMagazine]

G1:Maintain[QualityOfPublishedArticles]

G2:Maintain[OriginalityOfSubmission]

G3:Maintain[QualityofPrint]

G4:Achieve[EnoughQuantityOfPublishedArticles]

This tells us the root goal and the second level goals of the system. Further details about this specific system can be found in [1]

The following is the specification of the second level components of the tree.

Preskriptor Specification: ScientificPaperManager

KAOS Specification: PaperSelectionProcess

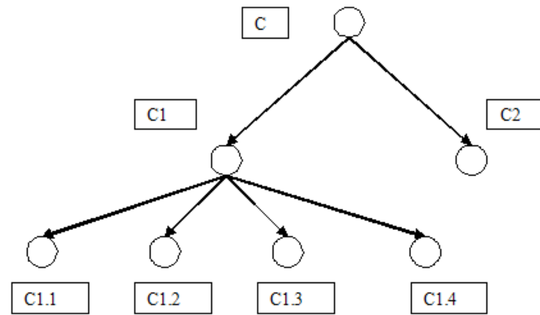

Figure 12: Component refinement tree for the paper selection process

Components:

Component SelectionManager

Type Processing

Constraints Maintain[QualityOfTheScientificMagazine]

Composed of /

Uses PeopleConnect to interact with (AutorAgent, ChiefEditorAgent, AssociatedEditorAgent, EvaluatorAgent)

Component PeopleConnect

Type Connector

Constraints Maintain[QualityOfTheScientificMagazine]

Composed of /

Uses /

Specifically for the paper selection process the second level resulting component tree has only two components. The first component is the SelectionManager which is the basic manager required to realize all the goals. PeopleConnect is the connector used by this manager to allow the various subsystems to interact.

A different approach is seen in the Power plant problem. In this case 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 case of the powerplant - 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 look 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

The Second Step

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 won't be a part of the system's prescription.

Since all the components derived from 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.

The Third Step

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 of 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 use different approaches.

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 creating sub components for Communication component was a step that helped illustrate these differences. Therefore we created the sub 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 /

This architecture does specify 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 connector description.

Achieving Non Functional Requirements

Different styles and patterns discussed in this thesis are used to perform transformation on the architecture to achieve various non functional requirements.

Box Diagram

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 helps in visualizing this and thus gives a more complete view of the architecture. Whereas the component tree gave us a hierarchical type view of the system, the box diagram gives us a network type view. This is essential in understanding how the system works. This diagram also helps in understanding the connectors of the system because it tells us the way these connectors link to components. This thereby helps 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.

Bibliography

[1] Manuel Brandozzi. From goal oriented requirements specifications to architectural prescriptions. Master's thesis, The University of Texas at Austin, 2001.

[2] Manuel Brandozzi and Dewayne E. Perry. Transforming goal oriented requirement specifications into architectural prescriptions. In Castro and Kramer, editors, *STRAW 2001 - From Software Requirements to Architectures*, pages 54–60, 2001.

[3] Manuel Brandozzi and Dewayne E. Perry. Architectural prescriptions for dependable systems. In *ICSE 2002 - International Workshop on Architecting Dependable Systems*, Orlando, May 2002.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture*, chapter 3. Wiley, 1996.

[5] Alberto Coen-Porisini and Dino Mandrioli. Using trio for designing a corba-based application. *Concurrency: Practical and Experience*, 12(10):981–1015, August 2000.

[6] Alberto Coen-Porisini, Matteo Pradella, Matteo Rossi, and Dino Mandrioli. A formal approach for designing corba based applications. In *ICSE*

2000 - 22nd International Conference on on Software Engineering, pages 188–197, Limerick, June 2000. ACM Press.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*, chapter 5. Addison Wesley, 1995.

[8] Emmanuel Letier and Axel van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *ICSE 2002 - 24th International Conference of Sofware Engineering*, pages 83–93, Orlando, May 2002. ACM Press.

[9] Emmanuel Letier and Axel van Lamsweerde. Deriving operational software specifications from system goals. In *FSE-10 - 10th ACM Symposium on the Foundations of Sofware Engineering*, pages 119–128, Charleston, November 2002. ACM Press.

[10] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*, chapter 3. Springer-Verlag, 1992.

[11] Philippe Massonet and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *FSE-4 - 4th ACM Symposium on the Foundations of Sofware Engineering*, pages 179–190, San Fransisco, October 1996. ACM Press.

[12] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[13] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[14] Axel van Lamsweerde. From system goals to software architecture. In Marco Bernardo and Paola Inverardi, editors, *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer-Verlag, 2003.

Vita

Divya Jani was born in Jaipur, India on December 16, 1980, the daughter of S.P. Jani and Pravina Jani. See received the Bachelor of Science degree in Engineering from Rutgers University. Through the course of her studies she interned at Lucent Technologies and Dell Inc. After finishing her bachelors degree she applied to the University of Texas at Austin for enrollment in their computer engineering program. She was accepted and started graduate studies in September, 2002.

Permanent address: 4404 E Oltorf Apt 13203
Austin, Texas 78741

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.