UNIVERSITE CATHOLIQUE DE LOUVAIN
Faculté des Sciences Appliquées
Département d'ingénierie informatique

Deriving architectural descriptions from goal-oriented requirements

Promoteur : Prof. Axel van Lamsweerde
Co-promoteur : Prof. Dewayne E. Perry

Mémoire présenté en vue de l'obtention du grade d'Ingénieur Civil en Informatique

par
Damien Vanderveken

Louvain-la-Neuve
Année académique 2003-2004

Abstract

Requirements and software architecture form two essential steps of the software life cycle. Although closely inter-related, most research has considered them to date as isolated products. On the one hand methodologies have been developed in order to gather complete, coherent, consistent and adequate requirements. On the other hand work on architecture has focused on enabling precise descriptions and on allowing formal reasoning about system properties such as deadlock, starvation, race conditions and so on. However, the problem of building an architecture that actually satisfies the requirements is so far largely unaddressed. Nevertheless two methods have recently emerged in this direction: the KAOS method and the Preskriptor process. Both are still at an early development stage and need therefore a validation to be said effective.

This thesis validates those two approaches on a case study of significant size (a power plant supervisory system built for ENEL, the Italian electricity company). It results from the analysis that both methods seem to be productive. Indeed both resulting architectures satisfy all the functional requirements and most of the non-functional ones. Some weak points were also identified among which the absence of behavioral aspects in resulting architecture descriptions and weak pattern descriptions.

This work extends the KAOS method by proposing the use of an architecture description language to complete and precise the architectural description. It results from there a set of rules and heuristics enabling to derive a structural and behavioral architecture description. In addition two patterns are described in details, one to achieve fault-tolerant communication and the other to maintain data consistency. Both the resulting architecture fragment and the implied transformation process are discussed.

Acknowledgments

Foremost I thank my supervisor, Axel van Lamsweerde, to have introduced me to Dewayne E. Perry, so giving me the opportunity to perform a five months stay in United States. Moreover I thank him for his precious feedback, his encouragements and support all along the year.

I thank Dewayne E. Perry for his warm welcome at the University of Texas at Austin, for his availability and his logistic support. I benefited from his wise advices and from his experience.

I also wish to thank Divya Jani since part of this work was realized in close interaction with her. It has been a pleasure to collaborate with her.

I am indebted to Christophe Ponsard for his helpful comments on earlier versions of this thesis. His suggestions significantly improved the quality of this work.

I also thank my two reviewers, Christophe Ponsard and Simon Brohez, for their interest about my work.

Finally, I wish to thank all my family for their every day support and more particularly my sister. Her help was invaluable.

Contents

Introduction **7**

1 Background **9**

1.1 Goal-oriented Requirements Engineering 9

1.1.1 Introduction 9

1.1.2 KAOS: A Goal-Oriented Requirement Engineering Method 10

1.2 Architecture Description Languages 18

1.2.1 Software Architecture 18

1.2.2 ADLs 18

1.3 From System Goals to Software Architecture 24

1.3.1 The KAOS Method 24

1.3.2 The Preskriptor Process 29

2 Architecture derivation for a Power Plant Supervisory System **34**

2.1 Informal Description of the Problem 34

2.2 Requirements Analysis 38

2.2.1 Requirements Elaboration 38

2.2.2 Obstacle Analysis 49

2.3 Architecture Derivation 56

2.3.1 Using the KAOS Method 56

2.3.2 Using the Preskriptor Process 61

2.3.3 Comparing the Resulting Architectures 68

2.4 Discussion 71

2.4.1 Evaluating the Methods 71

2.4.2 Opportunities for Improvements 75

2.4.3 Comparing the Methods 76

3 Toward More Precise Architecture Derivation 78
3.1 Wright 79
3.2 Deriving Architectures in Wright 81
3.2.1 Integration within the KAOS method 81
3.2.2 Structure 82
3.2.3 Behavior 84
3.2.4 Elaboration of Scenarios 95
3.3 Application to the Power Plant System 96
3.4 Making Architectural Patterns Further Precise 103
3.4.1 The Fault-Tolerant Communication Pattern 103
3.4.2 The Observer Pattern 113
3.5 Discussion 119

Conclusion 121

Bibliography 124

List of Figures 128

List of Tables 130

A KAOS Specifications 131
A.1 Goal specifications 131
A.1.1 Functional goals 131
A.1.2 Non-functional goals 142
A.2 Object Specifications 143
A.3 Agents Specifications 149
A.4 Operations specifications 154
A.5 Modifications resulting from the obstacle analysis 162

B Architectural Prescriptions 165
B.1 Initial Prescriptions 165
B.2 Prescriptions resulting from step 4 169

C Wright Specifications 172
C.1 The Fault-Tolerant Communication Pattern 172
C.1.1 Initial Wright Specification 172
C.1.2 Resulting Wright Specification 173
C.2 The Observer Pattern 176
C.2.1 Initial Wright Specification 176
C.2.2 Resulting Wright Specification 177

Introduction

Requirements and software architecture have long been recognized as two crucial parts of the software development process. Inadequate, inconsistent, incomplete or ambiguous requirements have a critical impact on the quality of the resulting software. So does software architecture. Its influence is particularly significant on non-functional properties such as performance, reliability, and security. Since ten years, the scientific community has therefore focused its efforts on the development of methods and tools so as to improve practices in those two areas. Therefrom goal-oriented requirement engineering and architectural description languages (ADL) have emerged as the most satisfactory solutions.

Far from being independent, requirements and architecture are closely inter-related. Requirements should serve as the basis to the architecture design while architecture must satisfy requirements. However, research has only recently considered the problem of building an architecture that satisfies the requirements.

The KAOS method and the Preskriptor process are two different approaches toward this direction. They both use goal-oriented requirements expressed in linear temporal logic using the KAOS framework in order to build an architecture satisfying functional and non-functional requirements. The KAOS method starts by deriving an abstract dataflow architecture in which all functional goals hold. It refines it further by applying styles and patterns in order to achieve architectural constraints and non-functional goals respectively. The Preskriptor process constructs an architectural prescription of the system. A component refinement tree is elicited from requirements so that each goal is ensured by a component.

However these two methodologies are still at an early development stage. They lack of a validation on real examples to prove their efficiency. The aim of this work is twofold; it is first to evaluate and compare both methods by applying them on a system of reasonable size and secondly to improve a method on the basis of weak points identified during the methods applica-

tion.

The case study deals with a power plant supervisory system developed for ENEL, the Italian electricity company. System description was extracted from various papers reporting this industrial experience. Therefrom the requirements specifications needed as starting point for both architecture derivation techniques were extracted. The KAOS method and the Preskriptor process were then applied to derive the software architecture.

The main result which has emerged from the experiment analysis was the two methods seem to be effective. Indeed both resulting architectures ensure all functional and most of the non-functional requirements.

Those methodologies were nevertheless not perfect. The absence of behavioral description in the resulting architectures was identified as their main common weakness. Moreover patterns descriptions were also insufficient in the KAOS method.

An Architectural Description Language (ADL) was used to improve the KAOS method with respect to the identified problems. It results from there a set of rules and heuristics enabling to derive a structural and behavioral architecture description as well as a precise description of two patterns, one to achieve fault-tolerant communication and the other to maintain data consistency.

This thesis is structured as follows: Chapter 1 provides the necessary background material, Chapter 2 describes the case study and Chapter 3 explores the use of an ADL in order to add a behavioral aspect to architecture description and defines precisely two patterns. Finally the conclusion summarizes the main points of this work and presents some further work. The appendices include complete specifications of the requirements and of the derived architecture for the power plant supervisory system.

Chapter 1

Background

1.1 Goal-oriented Requirements Engineering

1.1.1 Introduction

Before being able to construct any system, it has first to be understood. *Requirements* of the envisioned system have to be defined. Its objectives together with the needs they fulfill state *why* the system is needed. Its features, be they functional or non-functional, state *what* the system has to do. Functional aspects consist of the services to be provided while non-functional ones deal with the quality of the developed software. They encompass safety, security, usability, flexibility, performance, robustness, interoperability, cost, maintainability, and so on. The way those requirements will be met state *how* the system will be built. Precise requirements should answer the WHY/WHAT/HOW questions.

Requirement engineering is concerned with the elaboration of such requirements. Goals to be achieved have to be identified then operationalized into services and constraints. The responsibility of achieving those two must next be assigned to some agents, be they part of the environment (e.g., human or devices) or of the software itself. One should note that requirements are not fixed once and for all. They need continuous review and revision. So requirement engineering is an iterative process

Requirement engineering is recognized as a stage of prime importance in the software development process. Inadequate, inconsistent, incomplete or ambiguous requirements have a critical impact on the quality of the resulting software. Studies have shown that a requirement error corrected at a late development phase could cost up to 200 times more than if it had been corrected during the requirement engineering phase. Poor requirements are

also a major cause of software failure according to the managers in charge of the project. By software failure, it is meant a project that was never completed or only partially.

Goals are an essential component in the requirement engineering process. They provide the rationale of the system answering the WHY question. Object-oriented analysis techniques do not address such concerns. This has led to a migration from an object orientation to a goal orientation.

Goal-oriented requirement engineering encompasses a set of techniques including goal modeling, goal specification, and goal-oriented reasoning. Two complementary approaches have emerged into two different frameworks: a formal and a qualitative one.

The *formal* framework assigns linear temporal logic formulas to goals and uses AND/OR refinements to structure them. Roughly, when a goal is AND-refined into subgoals it means that the satisfaction of all of its subgoals is a sufficient condition to satisfy this goal. Similarly, when a goal is OR-refined it means that the satisfaction of one of its subgoals is a sufficient condition to satisfy the goal.

In the *qualitative* framework, weaker versions of those links are introduced to relate "soft" goals. The links express contribution, be it positive or negative, from a goal to another. "Soft" goals denote goals whose satisfaction is difficult to check in a clear-cut sense. The concept of satisficing is introduced to express that some goal is achieved within acceptable limits, rather than absolutely. If a goal is AND-decomposed into subgoals and all subgoals are satisficed, assuming all the subgoals contribute positively, then the goal is satisficeable; but if a subgoal is denied then the goal is deniable.

1.1.2 KAOS: A Goal-Oriented Requirement Engineering Method

The *formal* framework gave rise to the KAOS methodology for eliciting, specifying and analyzing goals, requirements, scenarios, and responsibility assignments. This is the methodology used to elaborate the requirements of the power plant supervisory system.

Four complementary and interdependent models form the requirements in the KAOS method: (1) the goal model, (2) the object model, (3) the agent model and (4) the operation model. Each of them presents a different view of the system and consists of a graphical and a textual representation. KAOS has a two-level semantic. A semantic net layer captures goals, constraints, agents, objects and actions together with their link while the formal assertion layer uses real-time first order linear temporal logic [22] to support formal

specification and reasoning.

The *goal model* presents an intentional view of the studied system. Functional and non-functional goals are structured by AND/OR goal diagrams. Higher level goals are rather general and involve multiple agents while lower levels are more technical and involve less agents. Goals belong to various categories (e.g., Security, Information), can be of different types (e.g., Maintain/Avoid, Achieve/Cease) and are characterized by attributes (e.g., Name, Definition). Terminal goals are distinguished according to the agent they are assigned to. A *requirement* is a terminal goal assigned to an agent of the software-to-be while an *expectation* is a terminal goal assigned to an agent part of the environment. The latter cannot be enforced by the software-to-be. The refinement ends up when each leaf goal is *realizable* by a single agent.

The *object model* provides a structural view. Domain objects of interest are modeled by UML class diagrams. Objects can be entities, associations, events or agents and are characterized by attributes and invariants. Invariants can be *domain properties*, that is, properties about object of the environment that hold independently of the software-to-be. Objects reflect the state of the system at a certain point in time.

The *agent model* points out the responsibilities in the system. Agents are active components that play some role toward goal satisfaction. They can be either part of the software or part of the environment (e.g., humans, sensors, actuators). Goals under the responsibility of software agents are requirements while those under the responsibility of environment agents are expectations. Their capabilities in terms of monitored and controlled variables are expressed through context diagrams.

The *operation model* reveals the behavior of the system. Operations express state transitions over objects of the system. An operation is specified by the classic pre/postcondition mechanism. A distinction is though made between domain pre/post conditions and pre-, post- and trigger conditions required for the satisfaction of some goal. The dynamic can be expressed graphically using scenarios and state charts.

These four models are strongly interwoven. The definition of goals refers to objects. Agents perform operations that operationalize goals, i.e., ensure goal satisfaction. The interface of agents refers to object definition. Operation application defines a state transition of some object. The links between models ensure a coherent, consistent, complete, and adequate picture of the system is constructed.

Now that basic underlying concepts have been exposed, the KAOS methodology itself will be illustrated with simplified excerpts of requirements

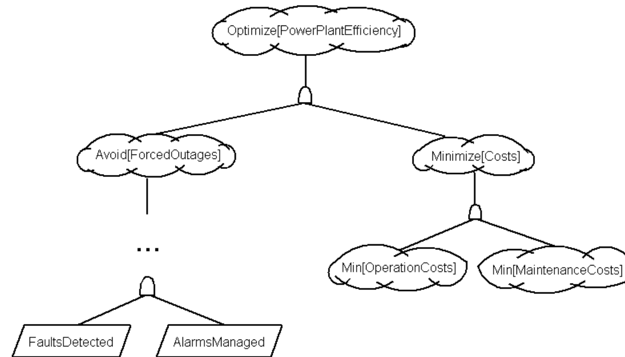

Figure 1.1: Preliminary goal graph for the power plant supervisory system

specification of the power plant supervisory system. A complete description will be exposed in section 2.2.

Goal identification from the initial document

A first set of goals are identified from the available sources [8, 9, 10] by searching for intentional keywords such as ”purpose”, ”objective”, ”concern”, ”intent”, ”in order to”, and so forth. Goals elicited during that step tend to be higher level and may consequently be unformalizable (soft goals). Figure 1.1 shows the first goals derived. Clouds denote soft goals while parallelograms denote formalizable goals. The classic graphic representation of AND and OR gate has been borrowed from digital circuit theory to represent AND and OR refinements.

Formalizing goals and identifying objects

This step consists in formalizing existing goals and in identifying objects, associations and attributes appearing in the goal specification. The goal `FaultDetected` for example may be defined precisely:

Goal `FaultDetected`

> **InformalDef** Faults occurring in any location (i.e., the steam condenser or the cooling circuit) must be detected. Faults occur in only one location at a time.
>
> **FormalDef** $\forall$ f: Fault, $\exists!$ l: Location:
> Occurs(f,l) $\Rightarrow \Diamond$ Detected(f,l)

From the definition of that goal, several objects and relationships can be identified. The resulting portion of the object model is presented in Figure 1.2.

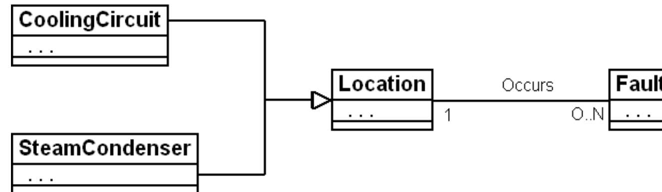

Figure 1.2: First draft of the object model

The goal `AlarmManaged` is hardly formalizable but an informal definition can nonetheless be given.

Goal `AlarmManaged`

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

The model previously built can now be enriched with the new concepts added by this definition (see Figure 1.3.

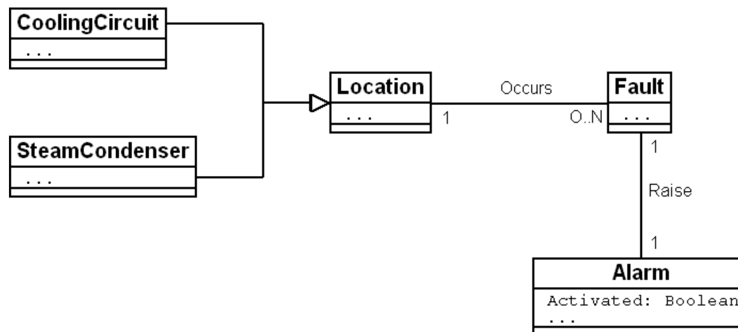

Figure 1.3: Second draft of the object model

Eliciting new goals through WHY questions

Finding more abstract goals, besides completing the goal diagram, can point out some subgoals missing in the first description. A lot of goals are often

implicit in the available sources and may be discovered as a by-product of the abstraction mechanism.

Abstraction translates an upward move into the graph and is achieved by asking the WHY question. Applied to the goals `FaultsDetected` and `AlarmManaged` it yields the parent goal `PowerPlantSupervised`. To achieve proper supervision, remedy actions should be suggested in case of fault detection. This leads to the originally missing subgoal `RemedyActions-SuggestedWhenFaultDetected`. The resulting subgraph is presented in Figure 1.4.

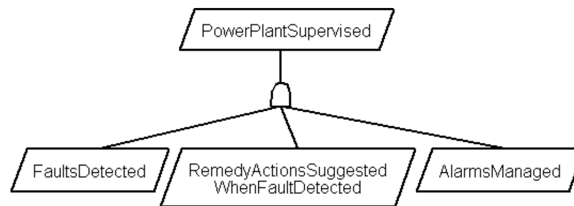

Figure 1.4: Addition of missing goals via WHY elicitation

Eliciting new goals through HOW questions

Goals have to be refined until leaf goals can be assigned to a single agent, be it part of the software or the environment. Goals should thereby be refined into more concrete ones.

Refinement translates a downward move into the graph and is achieved by asking the HOW question. Asking the HOW question to the goal `Alarms-Managed` leads to the two subgoals `AlarmRaisedWhenFaultDetected` and `AlarmsTraced`. The former can be further refined into `FaultInformationTransmittedWhenFaultDetected` and `AlarmRaisedWhenFaultInformationTransmitted`. Without going into too much details, the software agents in charge of faults detection and in charge of alarm management cannot be identical. This is why some information on diagnosed faults have to be transmitted. The resulting portion of the goal diagram is presented in Figure 1.5.

The refinement of `AlarmRaisedWhenFaultDetected` can be proved to be correct and complete using formal goal refinements patterns [12]. These patterns can also be used to derive automatically new refinements.

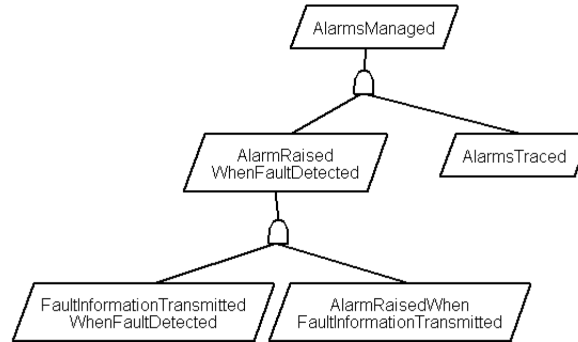

Figure 1.5: Addition of missing subgoals via HOW elicitation

Identifying potential responsibility assignments

During this step, terminal goals are examined and agents having the capabilities to take the responsibility of those goals are identified. Multiple agents can be able to achieve some goals. The goal `AlarmRaisedWhenFault-InformationTransmitted` could be assigned either to some automated (i.e., software or hardware) components or to a power plant employee. The former would raise the alarm automatically while the later would do it manually. The situation is presented in Figure 1.6.

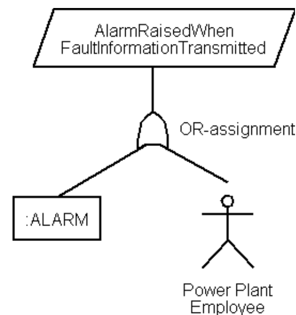

Figure 1.6: Potential agents

A qualitative reasoning can drive the selection among alternatives. In this case, assigning the goal to a power plant employee would contribute negatively to the soft goal `Minimize[OperationCosts]`. The software agent should therefore be preferred.

Deriving agent interfaces

Let assume that the goal `AlarmRaisedWhenFaultInformationTransmitted` has been assigned to a software component, namely `ALARM`. Its interface in terms of monitored and controlled variables can be derived from the formal specification of this goal:

Goal `AlarmRaisedWhenFaultInformationTransmitted`

> **FormalDef** ∀ fi: FaultInformation, ∃! a: alarm
> Transmitted(fi,PRECON,ALARM) ⇒ ◊ Raise(fi,a)

Note that `FaultInformation` is the software representation of the fault concept and that `PRECON` is the agent in charge of fault detection. `ALARM` needs consequently to be able to monitor `FaultInformation` and controls `Alarm`. Similar reasoning can be applied to other agents to derive their interface.

Identifying operation

For each goal, an operation is identified and preliminary defined by pre- and postconditions resulting from the domain. For the goal `AlarmRaisedWhenFaultInformationTransmitted` defined previously, the following operation can be defined:

Operation `RaiseAlarm`

> **Input** fi: FaultInformation, a: Alarm
>
> **Output** a: Alarm
>
> **DomPre** ¬Raise(fi,a)
>
> **DomPost** Raise(fi,a)

This operation definition only defines the basic state transition but does not ensure the goal.

Operationalizing goals

Once identified, operations have to be straighten in order to ensure the goal they *operationalize*. Required pre-, post- and trigger conditions are added to operations specification to ensure the satisfaction of goals. Operationalization can be automated using operationalization patterns [20]. In order to ensure the goal `AlarmRaisedWhenFaultInformationTransmitted` the specification of `RaiseAlarm` has to be modified:

Operation `RaiseAlarm`

Input fi: FaultInformation

Output a: Alarm

DomPre ¬Raise(fi,a)

DomPost Raise(fi,a)

ReqTrig for AlarmRaisedWhenFaultInformationTransmitted
@ Transmitted(fi,PRECON,ALARM)

The trigger condition states that when the trigger condition becomes true and if the preconditions hold then the operation must be applied.

The introduced KAOS methodology provides a set of both formal and semi-formal techniques enabling to build requirements. One should note that the method has been explained as being sequential but it was only for a presentation purpose. Far from the waterfall model, the KAOS method considers requirement elaboration as an iterative process.

In order to keep things concise, neither conflicts that can arise between goals nor obstacles to goal satisfaction have been addressed. However, these concerns are included in the method. More information on conflict management can be found in [28] and on obstacles handling in [29].

1.2 Architecture Description Languages

1.2.1 Software Architecture

Software architecture is concerned with the organization of software. It consists of a description of constituent architectural elements, their interactions and the constraints on those elements. Elements are divided into two categories: components and connectors. *Components* are used to denote points of computations while *connectors* define the interactions between these components. Interactions include communication and synchronization protocols which set up a coordination scheme between components. Typical constraints include security, performance or conformance to a particular style (e.g, pipes and filters, client-server, event-based).

Architecture can be viewed as a bridge between requirements and code and should thereby provide a framework in which to satisfy the requirements and serve as a basis for the design [24]. Architecture is besides known to have a major impact on non-functional requirements like performance, security, maintainability, interoperability.

When specifying an architecture, whatsoever be the mean used, some characteristics are desirable for the resulting description:

- Different aspects of the architecture have to be expressed in an appropriate manner. Aspects encompass structure (i.e., the way the different components and connectors are interconnected, the system topology), behavior (i.e., the abstract behavior of each component, the protocols of interactions they use to communicate and coordinate their effort in order to achieve some global behavior) and style (i.e., belonging to a particular style with its associated properties and constraints).
- Architecture descriptions should allow some form of reasoning to check whether or not some key system properties such as performance, reliability or security are satisfied and to what extent. In distributed environments, classical issues like presence of deadlock, starvation, race conditions should be addressed.

1.2.2 ADLs

Architectural description languages (ADL) are an attempt in this direction. They set up means enabling to describe and analyze architecture in a very much satisfactory way compared to traditional "box-and-line" architecture descriptions. Various formalisms allow precise specifications and formal reasoning. Their use is often supported by tools for displaying, compiling,

analyzing or simulating architectural descriptions. Lots of ADL have been designed over the last decade and they differ greatly by the intent pursued, the formalism used and the capabilities offered. Three representative ADLs[16] – C2 [23], Darwin [21, 18] and Wright [2, 1] – will be examined in details, looking for similarities and differences. Their capabilities with respect to the criteria stated above will be checked.

Although very different in the capabilities offered ADLs nonetheless share a set of primitive concepts that form the basis for any architecture description [16]. The main elements are:

- *Components* represent the primary computational and data stores of a system. Components often have interfaces describing any point of interaction between the component and its environment.

- *Connectors* represent interactions between components. They coordinate the linked components and are therefore often called the "glue" of the system.

- *Systems* are configurations of components and connectors. They capture the overall topology of the system. Some ADLs also provide hierarchical constructs, enabling some subsystems with their own internal structure to be described in terms of a new architecture.

- *Properties* express semantic information about a system and its components that goes beyond structure. For example, some ADLs allow to estimate the overall performance of a system expressed by a its latency time and throughput.

- *Constraints* prescribe claims about an architectural design that should remain true even if it evolves over time. One could imagine for example to constrain the topology of the system in such a way that all components are linked through binary connectors.

- *Styles* describe families of related systems. A style is typically described by a design vocabulary (i.e., the types of components and connectors), a composition rule (i.e., describing the allowable structure) and by invariants (i.e., properties that hold in any instantiation of the style) [26]. Typical examples of style include pipes and filters, layered systems, blackboard systems and client-server systems.

Comparative study

Now that the common basis has been provided, three considered ADLs are further described. In a first time it is important to introduce them and to reveal their aim. C2 is a component- and message-based style used to support the particular needs of applications that have a graphical user interface aspect. Darwin aims at modeling and analyzing distributed systems. Wright focuses on formal specifications and analysis of interactions between architectural components.

Modeling capabilities First ADL ability to model different aspects of architecture – structure, behavior and styles – is inspected.

Structure With respect to the structure description, the three ADLs take different approaches. C2 structures components into *layers*. Each component has a top and a bottom domain used to define the interface with the upper and the lower layer respectively. Layers are structured by level of abstraction, each layer n using services of component belonging to layer $n-1$. The principle of limited visibility holds between layers, that is, components of layer n are only aware of components belonging to layer $n-1$. In Darwin, a system consists of components linked through connectors. Each component defines a set of *services* provided and required. Connectors link services, one end provides the service the other end requires. A textual and a graphical representation support are used. Darwin allows composites components, that is, components that are constructed from basic or composite elements. So it enables to describe the system hierarchically. Wright distinguishes components and connectors as basic primitives. Each component defines its external interface by a set of *ports*, each one representing a particular point of interaction with the environment. Connectors are characterized by *roles*. Roles describe the expected behavior of the interacting parties. Connectors are then used to link components whose ports are *compatible* with the roles of the connectors. As in Darwin, hierarchical structures are supported. Structural aspects are handled in a more general manner in Darwin and Wright while the way C2 addresses structure reflects the intended audience. Actually, applications having a graphical user interface often decouple the presentation layer from the computation layer.

Behavior Behavior is as important as structure to describe architectures since it provides a mean to check whether the system will actually do what it is expected to. Surprisingly, the behavioral view is missing from

C2. Although not present in the early days, Darwin now supports behavior description of components using a finite process algebra (FSP). In Wright, the behavioral view is fundamental. As in Darwin, a process algebra is used, namely CSP in this case. Behavior of components *and* connectors can be specified. Each component possesses a computation field describing its global behavior and the ports describe its behavior at that particular point of interaction. Similarly connectors roles indicate the local expected behavior of the interacting parties while the *glue* field specifies how the activities of the different roles are coordinated. Compared to Darwin, Wright puts emphasis on connectors, adding a degree of freedom in the behavioral description while Darwin considers connectors as link elements without any role in the coordination of components.

Styles The increasingly recognized importance of styles in the software architecture field has made the ability to define a style in a ADL a “must”. Once again C2 does not support styles description. This is not strictly true since every architecture description in C2 follows implicitly a layered style and no emphasis is put on the resulting invariant of this style. Darwin does neither provide any constructs supporting style definitions. Only Wright really supports it. A style definition in Wright consists of descriptions of components and/or connectors, interface types and constraints. Component and connector can be parametrized in order to enable specialization for application purposes. Interface types, opposed to components and connectors, only constrain part of a component or connector. Constraints express properties that every style instantiation must obey. Such constraints include for example topological issues. Wright is here the only one that provides satisfactory style support.

Reasoning capabilities After evaluating the ADLs with respect to their abilities to express the different aspects of an architecture it is tried to examine the various forms of reasoning supported. It represents a move from a modeling point of view toward an analysis point of view.

C2 does support some analysis but not at the architectural level, strictly speaking. Type checking mechanisms are used to check whether a concrete component can be used as an implementation of an architectural component or whether a concrete component may substitute to another. Darwin provides both structural and behavioral checks. A configuration can be checked to see whether services provided and required by linked components are compatible. The behavior can be analyzed through the Labeled Transition

Analyzer (LTSA) tool. It enables to visualize execution traces and to check safety[1] and progress[2] properties. Similar checks are performed in Wright. The ports of components are compared to the roles of connectors to check whether they are behaviorally compatible. The analysis tools available for CSP – ProBE and FDR – can be used on Wright descriptions. ProBE[14] is an animator used to visualize potential executions while FDR[13] is a model checker enabling for example to check the presence of deadlock, starvation and race conditions. The level at which C2 performs its analysis deals with the dependencies between architecture and code phases. Darwin and Wright concentrate their analysis to the architectural level checking both structure and behavior.

Tool support The three presented ADLs are supported by tools. These tools belong to three categories:

1. Design Assistants: these tools focus on providing a graphical front end to allow architects to develop designs.
2. Design Checkers: they provide various analysis properties
3. Code Generators: they enable to generate code from the architectural description

C2 is supported by a design assistant and by a code generator. The code generator provides partial implementation that forms a development framework. Darwin has a development environment and as previously said design checkers. Wright only possesses design checkers.

Conclusion The evaluation of the three considered ADLs is summarized in Table 1.1. The results obtained bring the following conclusions. Wright is the more powerful. It is the only one to describe the three architectural aspects – structure, behavior and style and the analysis supported covers both the structural and the behavioral part of the architecture. Darwin is less complete in the architecture behavior description since no account is made of connectors. Moreover styles are not addressed. C2 seems more limited both in the description of the architecture and in the analysis provided. Nonetheless it points out an important fact: with the central position of architecture in the software engineering process between requirements and code, ADLs

[1] a *safety* property refers to a property that holds over all executions

[2] the *progress* property states that the program will eventually reach the desired state (it implies for example that the program never enters an infinite loop)

ADL	Structure	Behavior	Style	Reasoning	tools
C2	layered systems	no	no	implementation conformance	design assistant code generator
Darwin	all systems	components	yes	structure behavior	design assistant design checker
Wright	all systems	components connectors	yes	structure behavior	design checker

Table 1.1: Comparison between C2, Darwin and Wright

should provide some means to check that requirements are satisfied and the code complies with the architecture. Moreover none of the ADLs studied are able to perform analysis concerning critical non-functional properties such as performance, reliability, fault tolerance, security. This critic seems to apply to all ADLs in general.

1.3 From System Goals to Software Architecture

In Sections 1.1 and 1.2 it has been argued that requirements and architecture are essential in the software development process. Poor requirements are a major cause of software failure and architecture has a critical impact on non-functional requirements like performance, maintainability, security and so forth. These two products are inter-related. Requirements should serve as the basis to construct architecture while architecture has to satisfy requirements. The problem of building an architecture which satisfies the requirements is central to software engineering. This section expounds two architecture derivation methods with as starting point, KAOS goal-oriented requirements specifications. At this stage it is important to note there is a reduction of scope compared to requirements. Actually, requirements deal with both the software and its environment while the derived architecture concerns only the software itself. The first methodology has been developed by Axel van Lamsweerde [27] while the second one is the fruit of the work of Dewayne E. Perry and Manuel Brandozzi [4, 5, 6].

1.3.1 The KAOS Method

The derivation method consists of three incremental steps, each one achieving a particular purpose. The first stage builds a first architectural draft that satisfies functional requirements. This draft is globally refined during the second stage using styles to meet architectural constraints. The final architecture is eventually obtained by applying locally pattern refinements in order to ensure non-functional requirements.

From Software Specifications to Abstract Dataflow Architectures

This step aims at constructing an architectural draft which satisfies all functional requirements. Each software agent assigned to a functional requirement becomes a component. The data dependencies existing between software agents yield dataflow connectors between corresponding software components. Fine-grained components are preferred so as to achieve the non-functional soft goal `Maximize[Cohesion(C)]`.

More formally, the abstract dataflow architecture is derived from KAOS requirements specifications as follows:

1. For each functional goal assigned to the software-to-be, define one component regrouping the responsible agent together with the operations

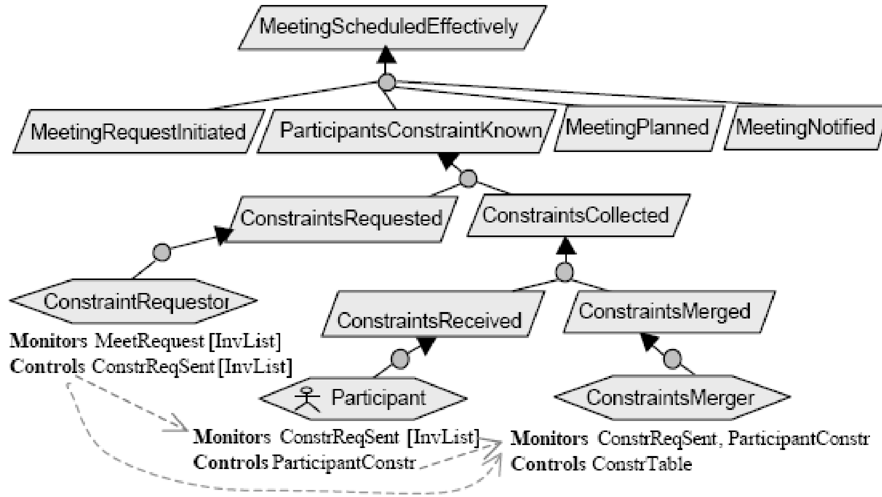

Figure 1.7: Assigned agents, their interfaces and data dependencies

operationalizing the goal. It is here that the distinction is made between the software and its environment since the only goals considered are those assigned to the software-to-be.

2. For each pair of components C1 and C2, derive a dataflow connector from C1 to C2 labeled with variable d iff d is among C1's controlled variables and C2's monitored variables:

$$Dataflow(d, C1, C2) \Leftrightarrow Controls(C1, d) \wedge Monitors(C2, d) \quad (1.1)$$

This step is illustrated on the "meeting scheduler" problem. Figure 1.7 shows a portion of the goal graph while Figure 1.8 exhibits the resulting dataflow architecture.

In the dataflow architecture, each component is characterized by the specifications of the goal assigned to it together with the specifications of the operations operationalizing that goal.

Since every functional goal has been assigned to a software component performing the necessary operations to ensure that goal, one can be convinced that all the functional requirements are satisfied.

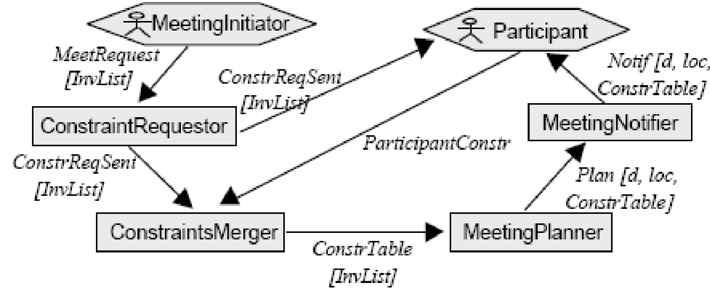

Figure 1.8: Derived dataflow architecture

Style-based architecture refinement to meet architectural constraints

The abstract dataflow architecture obtained in Section 1.3.1 defines our refinement space. This space may need to be globally constrained by architectural requirements. These typically arise from domain-specific features of environment agents or relationships among them, e.g., the distribution of human agents, organizational data or physical devices the software is controlling. An other possible cause could be the need for the software to integrate into a pre-existing system.

In this step, the architectural draft obtained from step 1 is refined by imposing a "suitable" style, that is, a style whose underlying goal matches the architectural constraint.

Style-based refinements are expressed through transformation rules. The advantage is twofold: first it documents style by applicability conditions[3] and effect conditions, secondly it makes the step more systematic.

An example of transformation rule for the *event-based* style is presented in Figure 1.9. The "house" notation is used to denote a domain property. Standard arrows denote dataflow connectors. A grey dashed arrow labeled by `?d` means that source component *registers interest* to the target component for events corresponding to productions of `d`. The latter events carry corresponding value for `d`.

The result of the application of the event-based style transformation rule (see Figure 1.9) on the abstract dataflow architecture of Figure 1.8 is shown on Figure 1.10.

[3]such as the architectural constraint and soft goals the style addresses

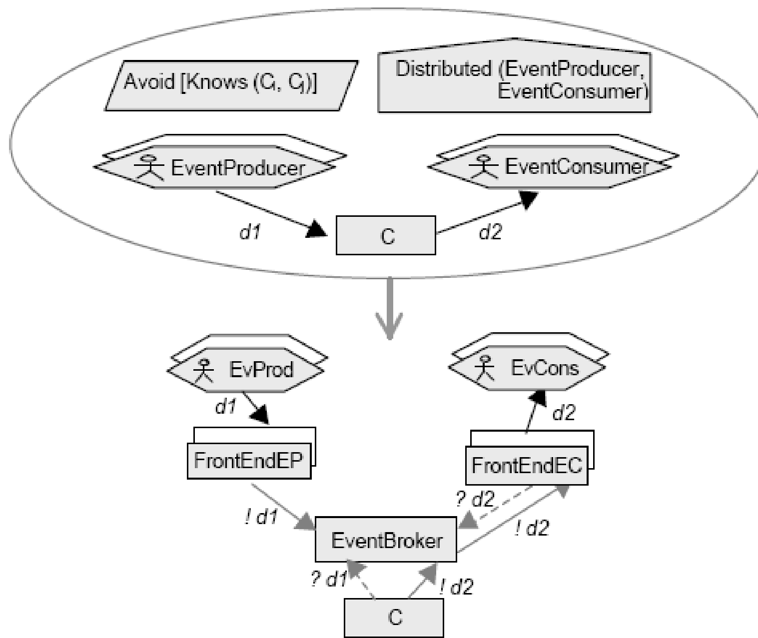

Figure 1.9: Event-based style transformation rule

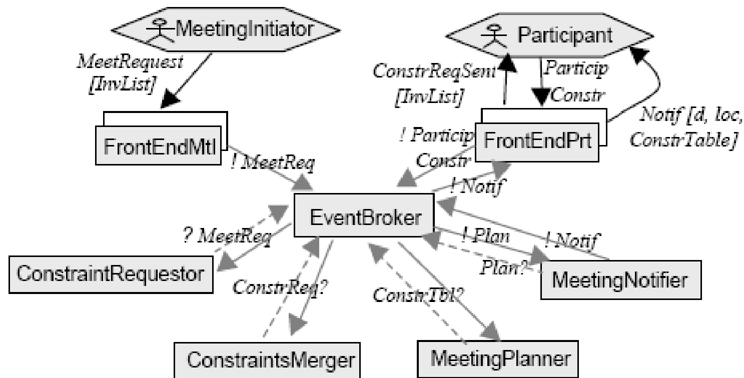

Figure 1.10: Style-based architecture

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-functional requirements. Those can belong to two different categories; they can be either quality-of-service or development goals. Quality-of-service goals include, among others, *security*, *accuracy* and *usability*. Development goals encompass desirable qualities of software such as *low coupling*, *high cohesion* and *reusability*. Many of these goals impose constraints either on components interactions or on single components.

This step refines the architecture in a more "local" way than the previous one. Patterns are consequently used instead of styles. The procedure to follow can be divided further into two intermediate steps.

1. for each non-functional goal (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 instantiated NFG to the constrained components. If more than one is applicable, select one using some qualitative technique (e.g., NFG prioritization).

Architectural refinement patterns are represented via rewriting rules consisting of a source architectural fragment, a target architectural fragment and a set of NFG achieved by the target. For example, the `NoReadUpNoWriteDown` pattern is shown in Figure 1.11. Note that the required postcondition of the component `SecurityFilter` is derived formally using operationalization patterns.

This third step is illustrated on the architecture obtained in Figure 1.10. First the impact of the confidentiality goal `Avoid[ParticipantConstraintsKnownToNonInitiatorParticipants]` is localized on the dataflow connector between the `MeetingPlanner` component and the `MeetingNotifier` component via the `EventBroker` component. Second the `NoReadUpNoWriteDown` pattern is identified as matching the confidentiality goal. Two disclosure levels are introduced: one for the meeting initiators and the other for normal participants. The instantiated pattern is then applied, resulting in the introduction of a new component – `ParticipantConstraintFilter` – between the `EventBroker` and the `MeetingNotifier`.

A sample of patterns achieving quality-of-service and development goals is presented in [27].

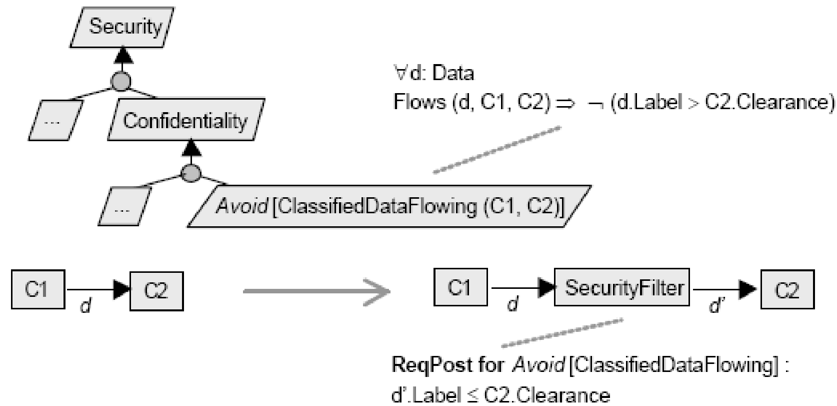

Figure 1.11: The `NoReadUpNoWriteDown` pattern for condidentiality goals

1.3.2 The Preskriptor Process

The second method converts the goal oriented requirement specifications of KAOS into architectural prescriptions. An architectural prescription is the architecture of the system in terms of its components, the constraints on them as well as the interrelationships among the components. Architectural prescriptions differ from classical descriptions by the fact that they are expressed at a higher level in the problem domain language rather than in the solution language. The intent is twofold: first it is to make the transition from requirements to architecture easier since there is a common vocabulary, secondly it is to favor reuse of pieces of architectural prescriptions since prescriptions are more general. This choice comes at a cost. Indeed prescriptions provide less guidance to designers than usual descriptions.

The components in an architecture prescription can be of three different types - process, data or connector. Processing components perform transformation on the data components. The data components contain the information to be used and transformed. The connector components encapsulate the various forms of interaction among process and data component, so holding the system together (the *glue*). All components are constrained by goals that they are responsible for. The interactions and restrictions of these components characterize the system. Here is an example of a component prescription expressed in the Architecture Prescription Language (APL):

Component Scheduler

Type Processing

Constraints MeetingScheduledEffectively,
MeetingRequestInitiated,
ParticipantsConstraintKnown,
...

Composed of PlanningEngine,
ParticipantClient,
ParticipantInitiatorClient,
RessourcesAvailableRepository,
SecureConnector1,
...

Uses /

This example shows a component called `Scheduler`. The *type* field denotes that the component is a processing type component. The *constraints* are the various goals realized by `Scheduler`. It thereby defines the constraints on the component. *Composed of* illustrates the sub components that implement `Scheduler` in the next refinement layer. The last attribute, *Uses*, indicates what are the components used by this component. It also specifies the connectors used for the interaction.

There exist some correspondences between KAOS entities and APL entities. Table 1.2 shows the potential mappings. It will serve as guidance during derivation process.

The Preskriptor process is composed by three required steps and by a fourth optional one. Figure 1.12 illustrates the interactions between the required steps.

Step 1: Derivation of the skeleton of the architecture

The first step considers the root goal of the requirements specification, and the other systems the software will have to interact with. From there a very high level description of the system is obtained. It is composed of a central processing element that achieves the root goal, and of a connector per surrounding system with which there will be some interaction.

The central processing element can be refined further in order to set up the skeleton of the component refinement tree. The structure of the goal diagram can serve as a basis.

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

Table 1.2: Mapping KAOS entities to APL entities[4]

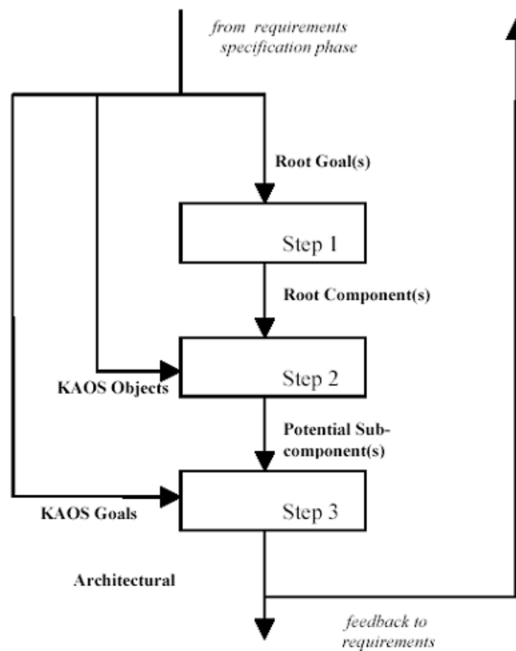

Figure 1.12: The 3 required steps of the Preskriptor process

Step 2: Identification of potential subcomponents from objects specification

Once the basic architecture is in place, potential subcomponents are derived from the objects in KAOS specification. The derived components can be either process, data or connectors. The selection among these possibilities is guided by the correspondences presented in Table 1.2. If in the third step, no constraint is assigned to these components, they won't be part of the final system prescription.

The derived components are then used to refine the architectural skeleton obtained in step 1. Different rationale lead to different decompositions.

Step 3: Selection among potential components via constraints assignment

In this step it is determined which of the sub goals are achieved by the system. They are then assigned as constraints to the previously defined components. A level of refinement in the goal diagram is selected and serves as reference. The choice of the level imposes the degree of constraint put on the resulting architecture. A high level gives more freedom to designers, possibly enabling innovative solutions but provides them little guidance. A low level may constrain too much the architecture but is more helpful. It is then decided which of the potential components of step two will take responsibilities of the various selected goals. Note that this is a design decision made by the architect, decision based on the way he or she has chosen to realize the system. The components without constraints are discarded, and it is ended up with the first complete prescription of the system.

The resulting system achieves all the goals, be they functional or non-functional, provided they belong to the goal diagram. A distinction is made in the non-functional goals between those coming from the domain and the others. For example the former would include security goals in a bank application while the later includes desirable properties not specific to the domain like reusability. The result of the three first steps is called a *problem oriented prescription* since the goals treated only come from the problem domain.

Step 4: Architecture refinement to achieve non-problem goals

The fourth step aims at refining further the architecture in order to achieve additional non-functional goals that are not from the domain. These goals

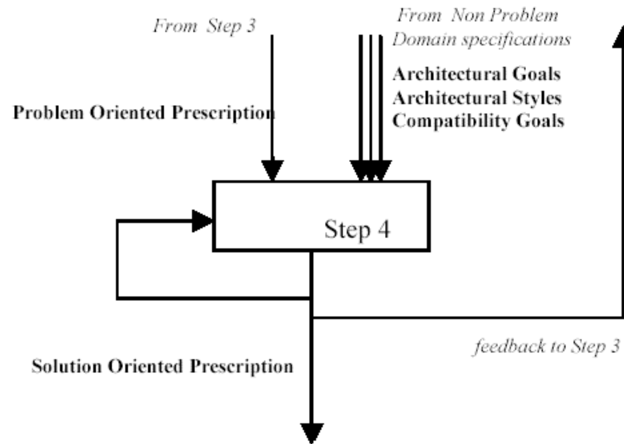

Figure 1.13: Step 4 of the Preskriptor process

are typically development goals (e.g., reusability, evolvability, maintainability), quality-of-service goals (e.g., performance, fault tolerance, usability), compatibility goals (e.g., CORBA) or conformance to a particular style.

The fourth step is iterated till all the non-domain goals are achieved. A graphical representation of this fourth step is presented in Figure 1.13. Given the problem oriented prescription of step 3, step 4 produces a *solution oriented prescription*. This terminology comes from the fact that the additional non-domain goals reflect needs for the particular developed product while the three first steps only concentrate on problem requirements.

Although the aim of this step is clear, its effective realization is still very vague. No general mean to achieve non-domain goals is given but specific examples are addressed. Reusability is achieved by splitting components [4], CORBA compatibility is added by constraining further some components [4] and fault tolerance is ensured by introducing a new connector and multiple copies of an existing component [6].

Chapter 2

Architecture derivation for a Power Plant Supervisory System

2.1 Informal Description of the Problem

This description results from the gathering of requirements pieces spread out in three different papers [8, 9, 10] in which Coen-Porisini et al reported their experience on an industrial project provided by ENEL, the Italian electrical company.

The application concerns an information system designed to support ENEL's personnel in managing thermal power plant operations. Power plants are typical examples of safety-critical systems since they may manipulate very hazardous substances in order to produce their electricity (e.g., fossil or nuclear fuel). Any error uncorrected could lead to critical failures in the power plant. The potential damage are huge in terms of human lives as well as for the environment. One can just remember what happened in Tchernobyl in 1986. Almost 20 years later the consequences of the nuclear reactor explosion are still killing people. The environment is polluted for hundreds of years. Reliability is therefore a main concern.

The purpose of the system is to optimize power plant efficiency, to reduce operating and maintenance costs, and to avoid forced outages by implementing (separately or in combination) functions related to supervision, condition monitoring, performance monitoring and fault diagnosis. Those functions include data acquisition from the field through sensors, detection of faults occurring in the power plant, and raising of appropriate alarms in case of

fault detection.

The main problem with such a system is that each of the above functions is usually developed separately, as a *standalone* application. Therefore, when installed on the plant, each application needs its own field data acquisition system, data processing unit, data storage system, and man-machine interface. This in turn results in obvious drawbacks in terms of higher implementation, installation, maintenance, and training costs, increased operational complexity, confusion and distraction of the system users, and possibly even incorrect and unsafe plant management. To solve this problem, ENEL is trying to integrate the aforementioned functions in a unified environment, called the *Advanced Supervisory System*.

Figure 2.1 depicts a view of the main components of the Advanced Supervisory System:

- *PRECON* is a diagnostic system whose goal is to continuously monitor the performances of the plant in order to detect faults in the steam condenser or in the cooling circuit. Moreover, PRECON supports the operators by suggesting remedy actions

- the *Alarm Management* unit traces and keeps the state of the alarms issued by all modules and manages operator interactions.

- the *Configuration Management* unit allows dynamic reconfiguration. It enables to have a general supervisory system that can easily be configured according to the peculiar needs of single plants (e.g., the amount of produced power, the type of plant component, etc.)

- the *Data Acquisition and Pre-processing* unit acquires data from the field.

- the *Global Plant Data Base (GPDB)* is a virtual database, consisting of a set of distributed – possibly heterogeneous – databases, that contains both static and dynamic data coming from the Acquisition subsystem, PRECON, the Configuration subsystem, and other components of the supervisory system.

- the *Communication Resource* supports and centralizes communications among the different components.

More precise statements define further the requirements (functional, performance, usability, fault-tolerance, etc.) that the different components must satisfy. Real-time constraints are ubiquitous given the safety-critical nature of power plant supervision.

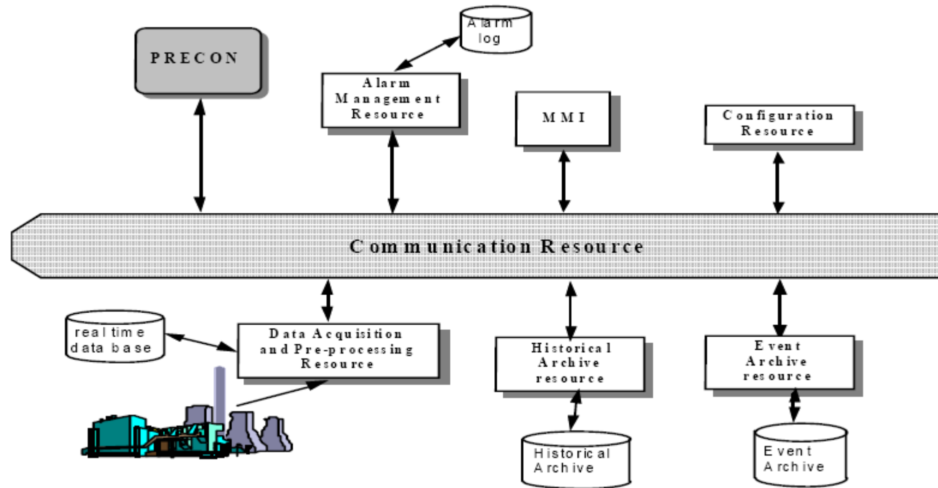

Figure 2.1: The ENEL's supervision and control system[8]

The Acquisition and Preprocessing resource must periodically acquire from plant sensors the measurements of analog and digital variables (mostly pressures and temperatures).

PRECON must perform its activity:

1. *periodically*, every 5 minutes; and
2. *upon user request*, whenever the user asks for some diagnosis. PRECON serves only one request at a time and must answer each of them within 5 seconds. The request service can be activated even when PRECON is performing its periodical diagnostic activity.

Each time PRECON achieves its task, it should thereafter store into the database the computed variables, the diagnosis of the current situation and the input/output data relation associated with the current situation detected.

The occurrence of an anomalous situation identified by PRECON must be notified to the Alarm Management that should raise the appropriate alarm. Alarms have to manage within an extremely short delay. Reliability is critical for the alarm management unit.

The Communication Resource should achieve strong time and reliability constraints.

The system must also tolerate some faults. There should be a system monitoring the activity of field devices (sensors, actuators, etc.) installed in

the power plant, in order to detect possible failures and malfunctions. Data collected from the field should be validated. To do so, physical devices can be asked to perform a self-test.

The system is subject to both hardware faults and disturbances which tend to change the value of a state variable, thus causing incorrect system operation. The system should be able, when a hardware fault occurs and remains unrepaired for at least *delta* seconds, to find the damaged part and put it off-line. The system operates normally only if the value of state variables is not altered by a disturbance.

2.2 Requirements Analysis

2.2.1 Requirements Elaboration

The requirements specifications were constructed using the KAOS method presented in Section 1.1. The informal description of Section 2.1 has provided the preliminary requirements assessment. However the description of the power plant supervisory system provided was partial and lacked details. So, throughout the requirement extraction process, it has been necessary to rely on personal engineering skills, on professor Perry's advices and on the common sense in order to gather requirements as realistic as possible. Note also that the reconfiguration function mentioned in Section 2.1 was not taken into account due to lack of time.

Since this thesis aims at deriving an architecture from the requirements rather than deriving requirements themselves, this section will be essentially descriptive rather than constructive especially as the KAOS methodology has already been explained on the power plant supervisory system in Section 1.1.2. Nevertheless a short paragraph will introduce for each model (i.e., goal, object, agent, and operation) some considerations on model elicitation. Next main characteristics of the model will be exposed.

The complete requirements specifications can be found in Appendix A. Appendix A should be used as reference companion. Moreover, the graphical representations of the different models are only present in the appendix. The reader is thus strongly advised to refer to Appendix A during the reading of this section.

Goal Model

Elicitation The following steps were followed in order to build the goal model. First of all, the informal definition of goals mentioned in [8] were carefully written down. From that, two first goal refinement trees were built, one for the functional goals and the other for the non-functional ones. Note that the soft goals are not addressed in this analysis since they are not really part of the KAOS constructs. This first draft was all but complete. These trees were completed thanks to a refinement/abstraction process. The version obtained at that point was still totally informal. Temporal first-order logic [22] was then used to remove this weakness. It enabled to ensure the refinement tree was correct, complete and coherent. The use of refinement patterns as described in [12] 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 Figure 2.2. The patterns were of a great help to track and correct incompleteness and incoherence. Furthermore they enabled to save a huge amount of time by avoiding to do the tedious proof work.

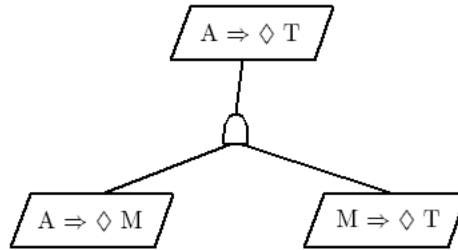

Figure 2.2: Milestone refinement pattern

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, omissions, etc.

Characteristics The goal refinement diagram representing the functional goals (see Figure A.1) is globally structured in two parts as presented in Figure 2.3. This shape reflects the two main goals the system has to ensure to monitor the power plant. The occurring 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 `AlarmCorrectly-Managed`. They are subsequently refined using the various patterns until the leaf goals are assignable to a single agent – be it part of the environment or of the software. The fault detection part deals with data acquisition from the field, the fault detection in itself, and the writing of a report after checks have been performed (see Figure 2.4). The alarm management part covers the raising of the appropriate alarm in case of fault detection, the management of information on alarms previously raised and interactions with a human operator (see Figure 2.5).

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

Goal `AlarmRaisedIfFaultDetected`

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

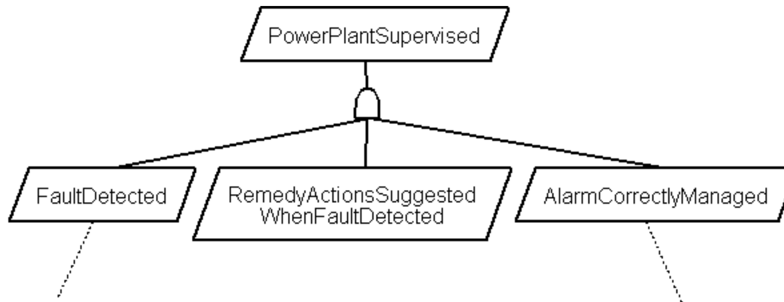

Figure 2.3: General Structure of the goal diagram

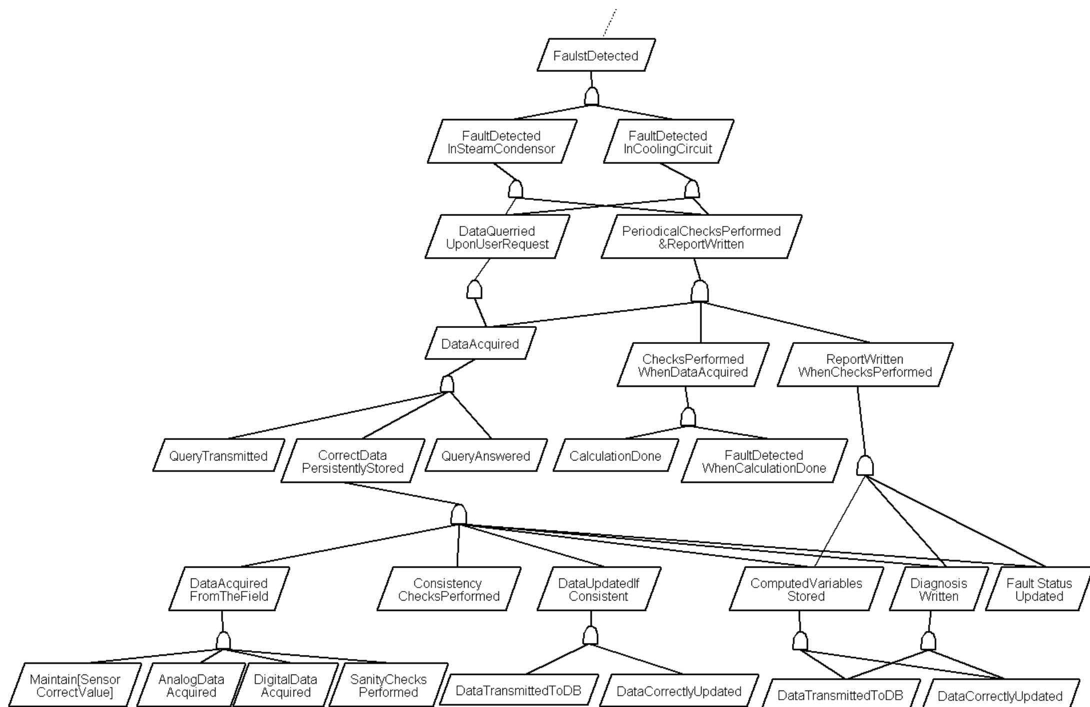

Figure 2.4: Refinement of the goal `FaultsDetected`

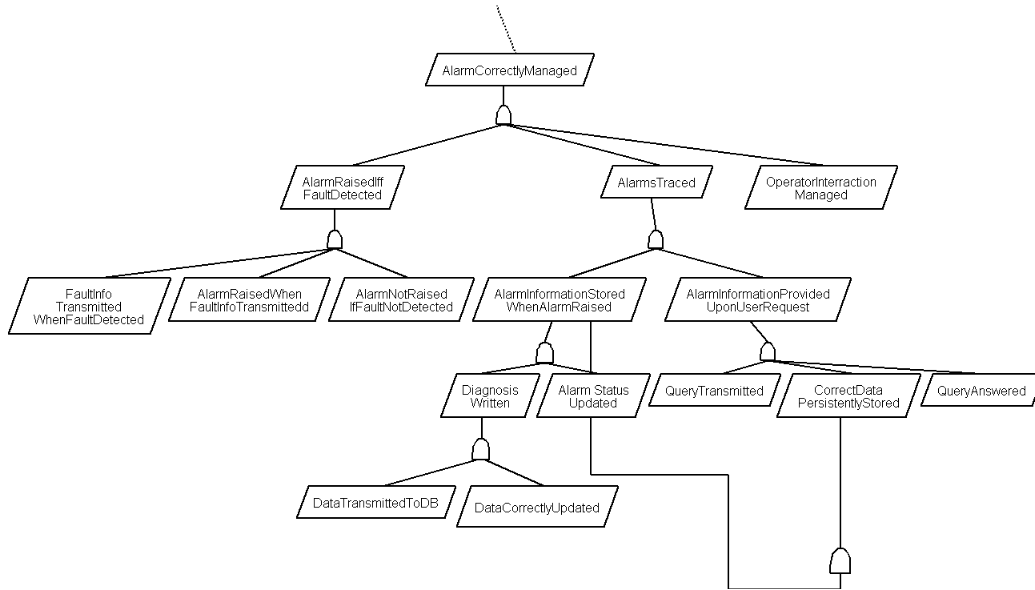

Figure 2.5: Refinement of the goal `AlarmCorrectlyManaged`

Detected(f,l) ⇒ ◊ (∃! a:Alarm, ∃! fi:FaultInformation) (Representation(fi,f) ∧ Raise(fi,a))

One can note that the presented milestone pattern does in fact only apply on propositional formulas. It can nonetheless serve as a guidance during goal refinement. Propositional reduction patterns can be generalized to first-order logic [11]. This goal is refined using as guidance the milestone refinement pattern (see Figure 2.2) into the following subgoals:

Goal `FaultInfoTransmittedWhenFaultDetected`

FormalDef ∀ f:Fault, ∃! l:Location
Detected(f,l) ⇒ ◊ (∃! fi:FaultInformation) (Representation(fi,f) ∧ Transmitted(fi,PRECON,ALARM))

Goal `AlarmRaisedWhenFaultInfoTransmitted`

FormalDef ∀ fi:FaultInformation
Transmitted(fi,PRECON,ALARM) ⇒ ◊ (∃! a:Alarm) Raise(fi,a)

The application of that particular pattern results from the fact that the information concerning the detected faults has to be transmitted to `ALARM` to

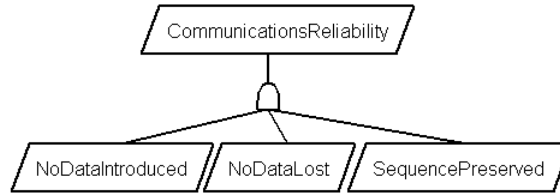

Figure 2.6: Communication reliability refinement subtree

enable it to raise the appropriate alarm. This intermediate state is necessary to reach the final state, i.e., the raising of the alarm.

In order to build a system as robust as possible, various goals have been added to the goal diagram.

Among these a first class takes care of the correct working of all the sensors and ensures the data provided is consistent and coherent. As described in Section 2.1, two types of actions are supported to do so: *consistency* and *sanity* checks. Once data have been acquired they have to be validated. This fact is represented by the goal `ConsistencyCheckPerformed`. Malfunctions of sensors – detected for example by asking suspicious sensors to perform self-tests – should result in switching their status to 'off', so putting it off-line. This is the purpose of `SanityCheckPerformed`.

The second category – represented by the goal `DataCorrectlyUpdated` – makes sure the updates are well performed by the database. The purpose of some goals is to maintain the power plant in a consistent state (e.g., `FaultStatusUpdated`, `AlarmStatusUpdated`).

The communication has also been constrained in order to prevent any transmission problems. This is expressed in the NFG diagram (see Figure A.2) through the *reliability* goals and the *performance* goal.

The refinement of the goal `CommunicationReliability` is the result of the robustness policy. The goal was refined as shown in Figure 2.6.

To model the data transmission, a relationship at the meta-level has been introduced. It is used to denote that some data has been transmitted from a sender agent to a receiver agent. Data can be either a whole object or only a part of it (i.e., a subset of its attributes). Figure 2.7 exhibits the transmission meta-relationship.

The three subgoals ensure the correctness of the transmission. They prescribe that no alteration has occurred on the data transmitted, that is, no data has been introduced or lost and the sequential order has been preserved. They have been formally refined as follows [1]:

[1]Data can either be `SensorInformation`, `FaultInformation`, `AlarmInformation`,

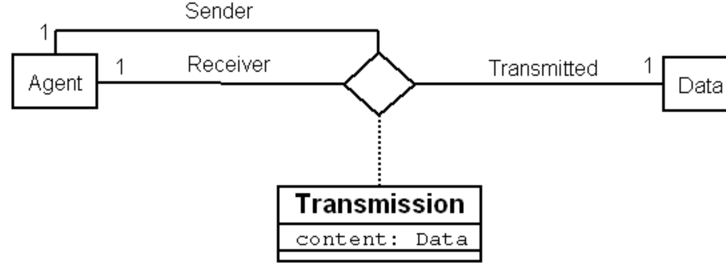

Figure 2.7: Transmission meta-relationship

Goal `NoDataIntroduced`

> **FormalDef** $\forall$ x:Data, X:$\mathcal{P}$(Data)[2], A_1,A_2:Agent
> Transmitted(X,A_1,A_2) $\wedge$ x $\in$ Transmitted(X,A_1,A_2).content
> $\Rightarrow$ x $\in$ X

Goal `NoDataLost`

> **FormalDef** $\forall$ x:Data, X:$\mathcal{P}$(Data),A_1,A_2:Agent
> x $\in$ X $\wedge$ Transmitted(X,A_1,A_2)
> $\Rightarrow$ x $\in$ Transmitted(X,A_1,A_2).content

Goal `SequencePreserved`

> **FormalDef** $\forall$ x,y: Data, X:$\mathcal{P}$(Data), A_1,A_2:Agent, $\exists$ u,v: Data
> x,y $\in$ X $\wedge$ Before (x,y,X) $\wedge$ Transmitted(X,A_1,A_2)
> $\Rightarrow$ u,v $\in$ Transmitted(X,A_1,A_2).content $\wedge$ x =u $\wedge$ y=v
> $\wedge$ Before(u,v,Transmitted(X,A_1,A_2))

Performance is also a major non-functional goal. Communication is constrained by a time bound. This is expressed by the NFG `Communication-Efficiency`. This limit varies throughout the system depending on the importance of the communication channel. The `FaultInformation` has to be transmitted from PRECON to ALARM within 1 second while answering a request can take a little longer – up to 5 seconds.

The formal definition of this goal depends on the time constraint. If one consider for example the transmission of a `FaultInformation` – which has the strongest time constraint – the formalization is a straightened version of the goal `FaultInfoTransmittedWhenFaultDetected`:

`FaultDiagnosis` or `AlarmDiagnosis`

[2]$\mathcal{P}$(Data) denotes the set of subsets of Data

Goal `CommunicationEfficiency`

FormalDef ∀ f:Fault, ∃! l:Location
Detected(f,l) ⇒ $\Diamond_{\leq 1s}$ (∃! fi:FaultInformation)(Representation(fi,f) ∧ Transmitted(fi,PRECON,ALARM))

Object Model

Elicication The object model constructed is a model of the composite system, that is, it models both real objects and their perception by the system. The motivation is to express accuracy goals. Figure A.3 shows the object diagram. Entities present in the objects were first derived from the informal definition of the goals. All the concepts of importance were modeled 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 only reflect a necessity. This necessity arises 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`.

Secondly the properties definition 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 ⇒ DeactivationTime = null

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 a lack of time. However it is believed that the only effect would be to modify the allowed range of temperatures and pressures. 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 in this case common sense. Among these are the attributes `Type` and `Power` of the object `PowerPlant`.

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

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 to manage the case where the representation of the object is not correct which would be unfortunate but could 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 two by two by a diagnosis relationship. The information provided by the sensor permits the detection of the faults and the description of a fault is the rationale for the raising of an alarm. Consequently the relationship `FaultDiagnosis` links `SensorInformation` and `FaultInformation` while `AlarmDiagnosis` links `FaultInformation` and `AlarmInformation`. Those two relationships are one-one. It is a modelling choice. It has been chosen 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.

Agent Model

Elicitation The definition of the agents was extracted mostly from [8, 9]. Inspiration has been drawn from the existing agents. Each leaf goal from the Goal Model was assigned to one of the agents. It was ensured that every agent has the capacity to assume the responsibility of the goal. By capacity it is meant 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 [19]. Figure A.5 shows the context diagram expressing controlled and monitored variables of each agent.

However a new agent was introduced : the `Instrumentation Maintenance System` (IMS). 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 different goals were assigned to the responsible agent. This step will be explained later in the Operation Model section.

Characteristics As already said, most of the agents come from the existing system. This is the case for `PRECON`, `ALARM`, `COMM`, `DB`, `Acquisition`

Unit and Sensor. The name used in [8] may be different but basically the performed functions 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 communication throughout the system. Moreover it performs all the transmissions between agents. DB persistently stores all the data and answers all the request concerning current values of the sensors, faults and alarms. The Sensor agent continuously measures the field variables and Acquisition Unit acquires data from the sensors. The resulting responsibility assignment is presented in Figure A.4.

The additional agent – IMS – has to check whether the sensors are working properly and if the data provided by the sensors are consistent in order to validate them.

The agents belong to two different categories; they can be either part of the software-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 agent are expectations while the others are requirements. This leads to the introduction of the IMS agent. Sensor is an environment agent and so all the goals assigned to it are expectations. But obviously it cannot be assumed that the goals SanityCheckPerformed and ConsistencyCheckPerformed will be true without the intervention of a reliable software device. Moreover those kind of tests should not be the responsibility of the Sensor from a conceptual point of view.

Operation Model

Elicitation 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. It is where all the difficulty lies: finding complete operationalizations. An extensive use of the operationalization patterns described in [20] has been made in order to derive complete operation specifications. It enabled to save a lot of time on proofs. It is even more true than for the goal refinement pattern as the application of the operationalization pattern has been found very systematic.

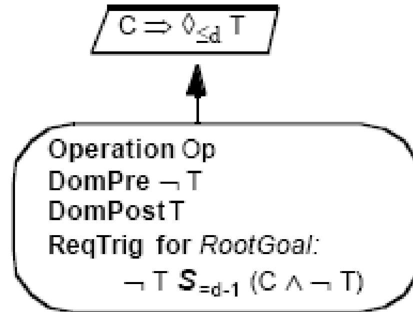

Figure 2.8: Bounded achieve opeartionalization pattern

Two patterns were particularly useful and have been used numerous times. The first one is the *bounded achieve* pattern described in Figure 2.8. Its applicability condition (i.e., $C \Rightarrow \Diamond_{\leq d} T$) makes it very popular. In fact most system goals have that form. The operation specification prescribes that $\neg T$ becomes T as soon as $C \wedge \neg T$ holds for $d - 1$ time units. It is then straightforward to see that such a specification operationalizes the goal $C \Rightarrow \Diamond_{\leq d} T$.

The second most useful pattern was the *immediate achieve* pattern described in Figure 2.9. Its applicability condition prescribes 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 guarantees the satisfaction of the goal. A intuitive explanation why will be given but the interested reader can find a complete proof in [20]. The first operation prescribes that as soon C becomes true the operation *must* be applied if $\neg T$ holds in order to reach the final state T. The second operation *may* be applied when C does not hold if the precondition T is true, making the postcondition $\neg T$ true.

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

Characteristics This section presents an illustration of the two operationalization patterns forementioned.

Concerning the first pattern, the operationalization of the goal `FaultInformationTransmittedWhenFaultDetected` will be examined. Its formal definition is given by

Goal `FaultInformationTransmittedWhenFaultDetected`

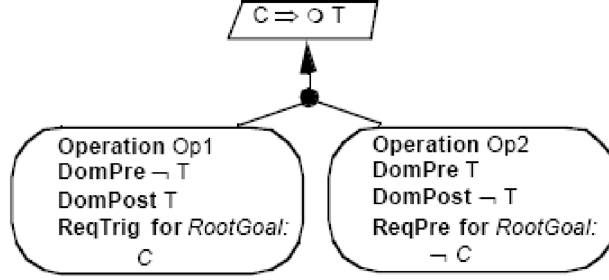

Figure 2.9: Immediate achieve operationalization pattern

FormalDef ∀ f:Fault, ∃! l:Location
Detected(f,l) ⇒ ◊ (∃! fi:FaultInformation) (Representation(fi,f) ∧ Transmitted(fi,PRECON,ALARM))

The pattern presented in Figure 2.8 can be instantiated using the following parameters:

C : Detected(f,l) ∧ Representation(fi,f)
T : Transmitted(fi,PRECON,ALARM)

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 ∧ Representation(fi,f) ∧ ¬ Transmitted(fi,PRECON,ALARM)

Note that as $d - 1$ time units makes here zero a smaller time unit is simply taken.

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

Goal `SanityCheckPerformed`

FormalDef ∀ s:Sensor
¬ s.workingProperly ∧ s.status='on' ⇒ ○ s.status='off'

The instantiation of the immediate achieve pattern presented in Figure 2.9 is straightforward.

C : ¬ s.workingProperly ∧ s.status='on'
T : s.status='off'

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

2.2.2 Obstacle Analysis

A recurrent problem with initial requirements specifications is that they tend to be too ideal. Unexpected behavior of agents like humans, devices, or software components can lead to violation of goals, requirements, or assumptions [29] resulting into differences between the system specifications and its actual behavior. Such differences can have a critical impact especially in safety-critical contexts like a power plant supervisory system. As the well-known proverb says: "Prevention is better than cure". The aim of this section is thus to anticipate the occurrence of undesirable behaviors from the system or from its environment in order to build more robust system specifications. The methodology used for this purpose is described in [29]. This is basically a two-steps process. First *obstacles* to goals satisfaction are identified. Secondly depending on various criteria (likelihood of the obstacle, importance of the obstructed goal, cost, etc.) strategies are applied in order to resolve the identified obstacles.

The obstacle identification and coverage presented here is not exhaustive. It has been focused on "critical goals", that is, goals thought as the most important with respect to the general purpose of the system.

Fault Detection

The first key function is the fault detection. `FaultDetectedWhenCalculationDone` is thus the first goal considered. Its formal definition is:

Goal `FaultDetectedWhenCalculationDone`

FormalDef $\forall$ f:Fault, l:Location
CalculationDone $\land$ Occurs(f,l) $\Rightarrow \Diamond$ Detected(f,l)
$\bigwedge$ CalculationDone $\land \neg$Occurs(f,l) $\Rightarrow \Box \neg$Detected(f,l)

Obstacles will be generated using *regression* for this first goal. For the analysis purpose, this goal will be further AND-refined into the two following subgoals:

(G1) $\forall$ f:Fault, l:Location
CalculationDone $\land$ Occurs(f,l) $\Rightarrow \Diamond$ Detected(f,l)
(G2) $\forall$ f:Fault, l:Location
CalculationDone $\land \neg$Occurs(f,l) $\Rightarrow \Box \neg$Detected(f,l)

The negation of the goals is given by:

(NG1) $\Diamond \exists$ f:Fault, l:Location
CalculationDone $\land$ Occurs(f,l) $\land \Box \neg$ Detected(f,l)
(NG2) $\Diamond \exists$ f:Fault, l:Location
CalculationDone $\land \neg$ Occurs(f,l) $\land \Diamond$ Detected(f,l)

The domain contains the two following property:

(D1) $\forall$ f:Fault, l:Location, $\exists$ s:Sensor
Detected(f,l) $\Rightarrow$ Monitors(s,l) $\land$ s.Status=on
$\land$ s.DataValue $\notin$ AllowedRange
(D2) $\forall$ f:Fault, l:Location, s:Sensor
$\neg$ Detected(f,l) $\Rightarrow \neg$ Monitors(s,l) $\lor$ s.Status=off
$\lor$ s.DataValue $\in$ AllowedRange

The property states that a necessary and sufficient condition for a fault detection is the presence of some working sensor monitoring the fault location with a value outside an allowed range. The second property states if a fault is not detected there cannot be a working sensor monitoring the fault location with a value outside the allowed range. These rules can be rewritten by contraposition:

(D1') ∃ f:Fault, l:Location, ∀ s:Sensor
¬ Monitors(s,l) ∨ s.Status=off ∨ s.DataValue ∈ AllowedRange
⇒ ¬ Detected(f,l)
(D2') ∃ f:Fault, l:Location, s:Sensor
Monitors(s,l) ∧ s.Status=on ∧ s.DataValue ∉ AllowedRange
⇒ Detected(f,l)

The right part of the equivalence in (D1') unifies with a literal in (NG1); regressing through (NG1) through (D1') then amounts to replacing in (NG1) the matching consequent in (D1) by the corresponding antecedent. Let's proceed similarly for (D2') and (NG2). The two following potential obstacles have thereby been formally derived:

(O1) ◊ ∃ f:Fault, l:Location, ∀ s:Sensor
CalculationDone ∧ Occurs(f,l)
∧ □ [¬ Monitors(s,l)∨ s.Status=off
∨ s.DataValue ∈ AllowedRange]
(02) ◊ ∃ f:Fault, l:Location, s:Sensor
CalculationDone ∧ ¬ Occurs(f,l)
∧ ◊ [Monitors(s,l) ∧ s.Status=on
∧ s.DataValue ∉ AllowedRange]

Obstacle 1 covers three situations, namely, the first where no sensor monitors the location where the fault occurs, the second where no sensor is working and the third where a fault effectively occurs while the value of the sensor is in the allowed range, the allowed range being thus not enough restrictive. Using OR-refinement three subobstacles could thereby be identified: `LocationNotMonitored`, `AllSensorOff` and `DetectionCriterion-NotStrictEnough`.

Obstacle 2 covers the situation where there is a working sensor monitoring some location whose value is outside the allowed range while no fault occurs in that location. This leads to the obstacle `DetectionCriterion-TooStrict`. The situation is summarized in Figure 2.10.

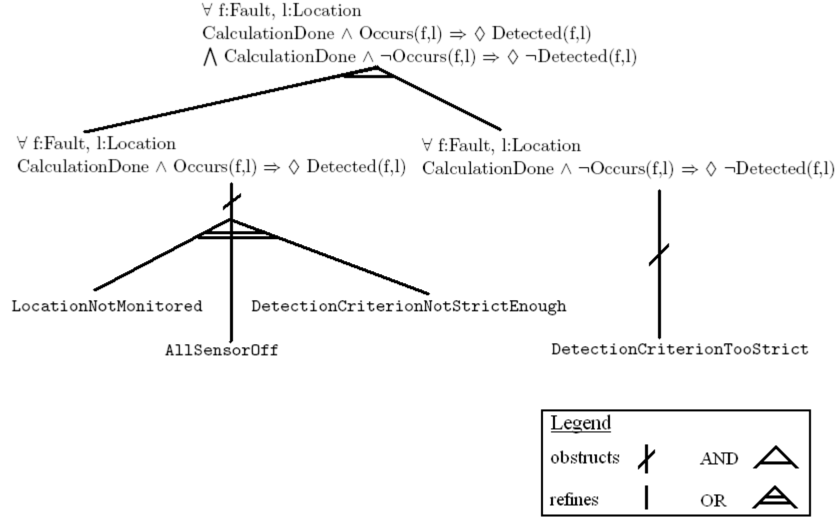

Figure 2.10: Refinement and obstacles for the goal `FaultDetectedWhen-CalculationDone`

Once obstacles have been identified, they have to be resolved. The obstacles `LocationNotMonitored` and `AllSensorOff` are resolved using the *Obstacle prevention* [29] by adding two new goals: `Avoid[LocationNot-Monitored]` and `Avoid[AllSensorOff]`. The case of `DetectionCriterion-NotStrictEnough` and `DetectionCriterionTooStrict` is far more complex. One should convince himself of the inherent difficulty of finding a good criterion with respect to fault detection. Having a criterion too restrictive leads to the detection of non-existing faults while having a criterion not restrictive enough leads to the non-detection of existing faults. To do a parallel with sensitivity test in statistics, straighten the criterion increases the number of false positives and loosen the criterion increases the number of false negatives. This is a critical aspect and the choice of an appropriate threshold should be considered very carefully. A strategy of goal mitigation could be imagined considering the fact that whatsoever be the chosen criterion, it will not be perfect. One reasonable solution would consist in adding some goal stating that if an error occurs in a fault detection, it will be corrected within a certain time bound. For simplicity and because the choice of such criterion is outside the scope of this thesis, it will be assumed in what follows that the chosen criterion is perfect.

Transmission of fault informations from `PRECON` to `ALARM`

Once a fault has been detected, information on the occurring fault have to be transmitted from `PRECON` to `ALARM` in order to raise the appropriate alarm. The goal considered here is `FaultInformationTransmittedWhen-FaultDetected`. Its definition[3] is given by:

Goal `FaultInformationTransmittedWhenFaultDetected`

FormalDef ∀ f:Fault, l:Location, ∃! fi:FaultInformation
Detected(f,l) ⇒ ◊ Transmitted(fi,PRECON,ALARM)
∧ Representation(fi,f)

The following domain property states that a necessary condition to the transmission of data from A_1 to A_2 is the correct operation of the source and the target of the transmission and of the communication channel (C). This property can be instantiated to `PRECON`, `ALARM` and `COMM` as follows:

(D) ∀ fi:FaultInformation
Transmitted(fi, PRECON, ALARM) ⇒ Operating(PRECON)
∧ Operating(ALARM) ∧ Operating(COMM)

Proceed similarly to what has been done for fault detection leads to the following obstacle:

(O) ◊ ∃ f:Fault, l:Location, ∀ fi:FaultInformation
Detected(f,l) ∧ □ [¬ Operating(PRECON)
∧ ¬ Operating(ALARM) ∧ ¬ Operating(COMM)
∧ ¬ Representation(fi,f)]

This obstacle mainly covers two situations, one where either PRECON, ALARM or COMM does not operate and the other where the transmitted fault information does not describe correctly the detected fault. Four subobstacles are thereby identified: `PRECONNotOperating`, `ALARMNotOperating`, `COMMNotOperating`, `IncorrectFaultInformationTransmitted`. One cannot afford to have one of these components down for whatever reason. Consequently, Fault-tolerant communication is added between `PRECON` and `ALARM` resulting in the addition of the new goal `Maintain[FaultTolerantCommunication(PRECON,ALARM)]`.

[3] A `FaultInformation` fi is the representation of a `Fault` f iff all their attributes are equal

The last obstacle corresponds to a classic problem where the representation in the software of an environment object is incorrect. This problem has been solved by adding the accuracy goal `Maintain[AccurateRepresentation(fi,f)]`. This problem can be extended to the whole set of software representations of environments objects and similar goals are consequently added for `SensorInformation` and `AlarmInformation`.

Data Acquisition

Since all the system relies on the data collected by the sensors, data acquisition is also a critical function. For brevity purpose, the entire obstacle analysis will not be exposed but one will rather concentrate on the key aspects emerging from such an analysis.

The first recurring obstacle is `InfoOutdated`. For example, data provided by a sensor can be diagnosed as inconsistent even though this is no longer the case just because sensor information is outdated. Moreover outdated sensor information value can lead to wrong fault detection when the actual sensor value is no longer outside the allowed range or to undetected faults when the fault occurs between two updates. The solution to that problem is to perform data acquisitions more often.

Surprisingly, updates can also be realized too quickly. An obstacle to the goal `SanityCheckPerformed` is the situation where the sensor is turned off while it is starting to work correctly again. Dually a sensor could be turned on while it is starting to behave incorrectly again. It is however very unlikely this situation takes place and the *Do-nothing* strategy is consequently applied.

A third problem can arise from the fact that sensor information are modified by two different agents, namely `Acquisition Unit` and `IMS`. Acquisition of data could take place while sensors are currently being turned off because of a malfunction. This would lead to undesirable inconsistencies. It has thereby been decided to add the goal `Maintain[AccurateData(Acquisition Unit,IMS)]` ensuring that both `Acquisition Unit` and `IMS` have accurate and coherent information on sensors.

Conclusion

The conducted obstacle analysis, although not complete, enabled to pinpoint some important unforeseen potential problems. The presence of a location not monitored, the absence of a working sensor, and an ill chosen detection criterion have been identified as obstacles to fault detection.

Inconsistencies between representations of environment objects and the objects themselves are general problem affecting the entire system. Non operating agents could be an obstacle to the correct operations of the system preventing critical functions such as fault detection, transmission of fault information from `PRECON` to `ALARM` and alarm raising. The issue of the time interval between two successive data acquisitions has been recognized as crucial. The potential incoherences resulting from the concurrent modifications on `SensorInformation` by `Sensor` and `ManagementUnit` agents have also been demonstrated.

Different strategies have been applied to solve the envisioned obstacles such as goal prevention, goal mitigation. Various goals have been added to the goal model in order to build a more robust system. Added goals are all non-functional and belong to different categories. `Avoid[LocationNotMonitored]` and `Avoid[AllSensorOff]`) are *safety* goals. `Maintain[FaultTolerantCommunication(PRECON,ALARM)]` is a *reliability* goal. `Maintain[AccurateRepresentation(fi,f)]` and `Maintain[AccurateData(AcquisitionUnit,IMS)]` are *accuracy* goals. Figure A.6 exhibits the non-functional requirements added by the obstacle analysis. One should also note the process of obstacle analysis can be iterative. In fact, the added goals could also be subject to various obstacles. However, an additional obstacle analysis has not been performed because the system is thought to be robust enough.

2.3 Architecture Derivation

This section presents the application of the two methods exposed in Sections 1.3.1 and 1.3.2 on the requirements elaborated in Section 2.2. For each methodology, the intermediate results and the derivation process will be commented. Afterward the two resulting architectures will be compared in details.

2.3.1 Using the KAOS Method

The architecture in this section will be derived using the method described in Section 1.3.1. The method prescribes the use of three different steps. The first step consists of the derivation of an abstract dataflow architecture from the KAOS specifications. This first draft is next refined using style in order to meet architectural 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.

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 modeled via dataflow connector. The followed procedure is divided into two sub-steps.

1. Each agent that assumes the responsibility of a functional 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:

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

This step is very systematic. From the context diagram shown in Figure A.5 it was needed to get rid off the environment agents to obtain the abstract dataflow architecture. The result is presented in Figure 2.11.

One can note certain features. Two components seem to be “absorbent”: `COMM` and `DB`. Four dataflow connectors come to each of them but none comes from them. This is caused by the fact neither `COMM` nor `DB` does control any

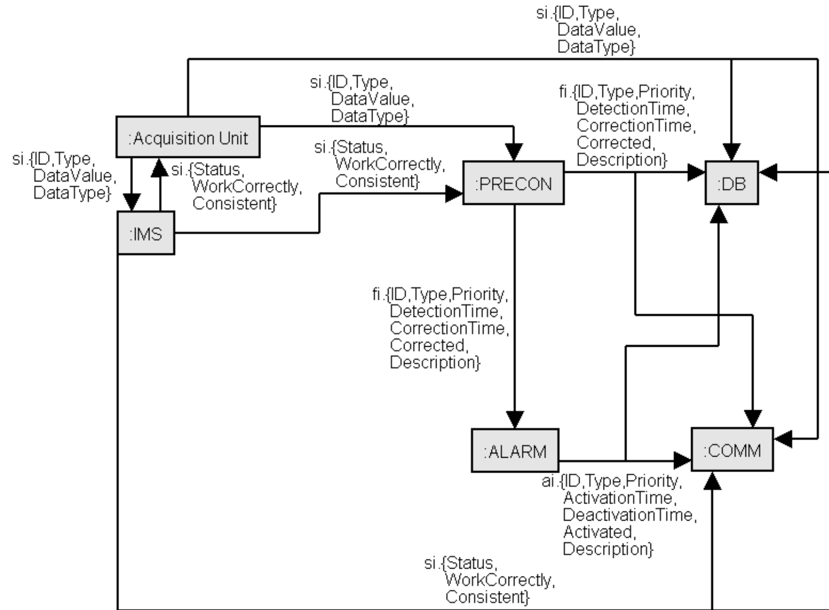

Figure 2.11: Abstract dataflow architecture

variables. In fact COMM carries all the data between the different components but does not perform any modification and DB stores all the data without any modification. Moreover there is a dataflow connector between PRECON and ALARM while the real data flow goes through COMM. A similar situation also happens between Acquisition Unit and PRECON. The real data flow should pass from DB through COMM but there is no dataflow derived.

It is believed 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. The resulting dataflow architecture may then appear quite strange.

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

In this step, the architectural draft obtained from step 1 is refined by imposing a "suitable" style, that is, a style whose underlying goal matches the architectural constraints. The main architectural constraint of this system [8, 9, 10] is that all the components should be distributed. In fact, in the real system, only PRECON has to be built and it has to integrate in a pre-

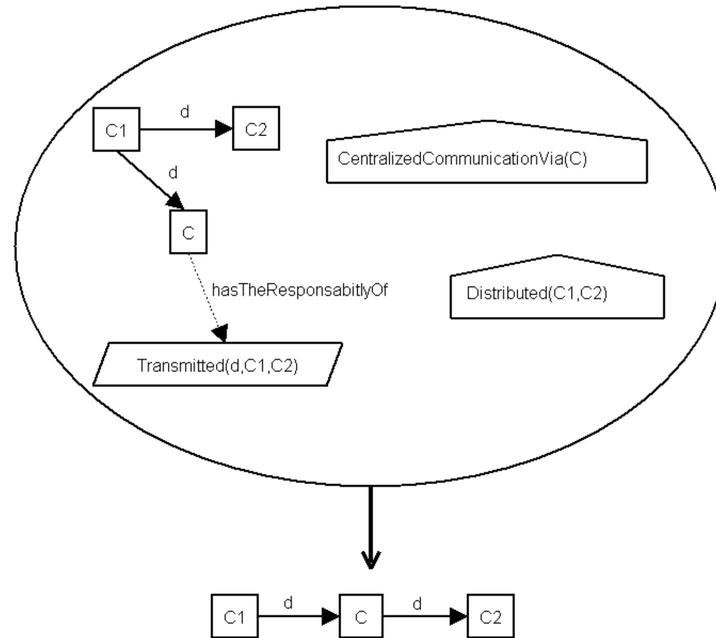

Figure 2.12: Centralized communication architectural style

existing architecture characterized by centralized communications and by distributed components.

The only transformation rule mentioned in [27] did not match this architectural constraints so a new one had to be designed. However, even if the transformation rule does not correspond, the produced architecture is centralized. The designed rule can therefore be seen as a variation of the event-based style where the transformation and the source architecture fragment are different while the target architecture fragment is similar. The `COMM` component plays in this case the role of broker. The resulting transformation rule is shown in Figure 2.12.

This style was applied on the architectural draft and the result is shown in Figure 2.13. The situation described by the rule arises 7 times, proof that the style constrains globally the architecture. Note that for presentation purpose, arrows coming from the `COMM` component to `DB` have been merged.

The architecture looks now closer to what was expected. Every single communication is achieved in a centralized way through the communication module. The architectural constraints are now met.

Nevertheless, one “absorbent” component subsists: `DB`.

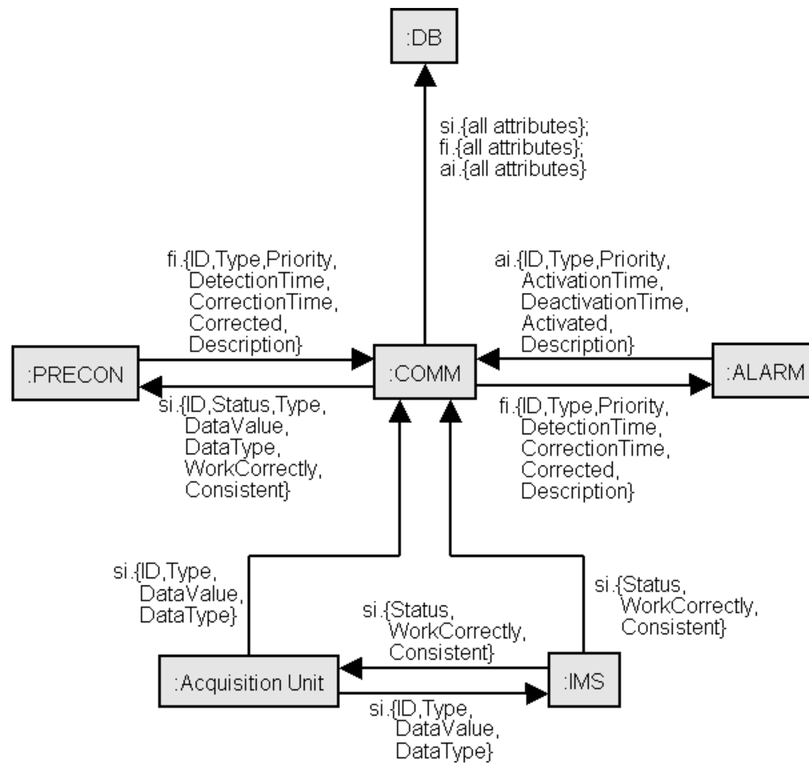

Figure 2.13: Style-based refined architecture

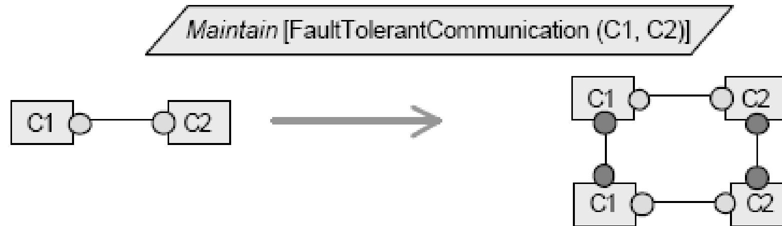

Figure 2.14: Fault-tolerant refinement pattern

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-functional requirements. Those can belong to two main categories; they can be either quality-of-service or development goals. Quality-of-service goals include, among others, *security*, *accuracy* and *usability*. Development goals encompass desirable qualities of software such as *low coupling*, *high cohesion* and *reusability*.

This step refines the architecture in a more local way than the previous one. Patterns are used instead of styles.

Two refinement patterns were used on the system. They come both from [27]. The first one is presented in Fig. 2.14. Thanks to the conducted obstacle analysis, the potential problems that could result from a misbehavior of either `PRECON` or `ALARM` have been pointed out. A fault-tolerant communication between those two components was required because they form the core of the system. They perform the most critical functions (i.e., the fault detection and the alarm management). This is the reason why those modules were desired to be as resistant as possible to any kinds of failure. The non-functional goal `Maintain[FaultTolerantCommunication(PRECON,ALARM)]` was therefore added. One could note than the pattern was not applied exactly according to its definition (see Figure 2.14). The presence of the component `COMM` between `PRECON` and `ALARM` was however ignored because it is believed it has no influence on the capacity of the pattern to achieve its goal.

The second refinement pattern used is shown in Figure 2.15. As the previous one, its introduction was motivated by the obstacle analysis. It pinpoints the potential incoherence and inconsistency problems that could arise since both `Acquisition Unit` and `IMS` access and modify the same data, namely information on sensor. The goal `Maintain[AccurateData(Ac-`

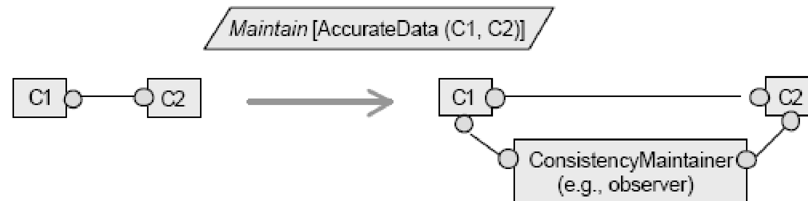

Figure 2.15: Consistency maintainer refinement pattern

`quisition Unit,IMS)]` was consequently introduced to make sure that all the modifications made from both sides lead to a consistent state.

The final architecture is presented in Figure 2.16.

2.3.2 Using the Preskriptor Process

The second method converts the goal oriented requirement specifications of KAOS into architectural prescriptions. The derivation process consists of four sequential steps as described in Section 1.3.2. The structure of this section will thus reflect this separation. The complete system prescriptions are presented in Appendix B.

Step 1: Derivation of the skeleton of the architecture

In the first step the basic prescription is derived 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. The root component is thus defined as `PowerPlantSupervisingSystem`.

This goal is then refined into `PRECON`, `ALARM`, `DataBase` and `Communication` components. This refinement is obtained by selecting a specific level of the goal refinement tree. Considering only the root of the goal refinement tree, the prescription would end up being too vague. On the other hand picking the leaves may end up with a too constrained prescription. Therefore the second level of the tree has been picked as it allows to create a very well defined prescription while preventing a specification constraining the lower level design. The second level enables to identify two components `PRECON` and `ALARM`. `Communication` was introduced thereafter because `PRECON` and `ALARM` were obviously *processing* component and needed thus a *connector* component to communicate. The constraint for the system to use central-

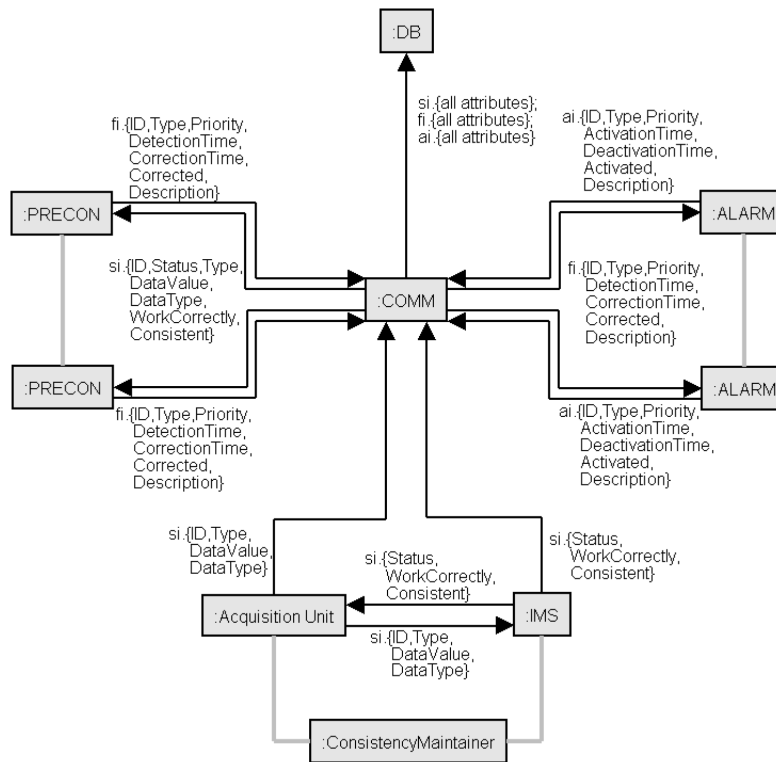

Figure 2.16: Pattern-based refined architecture

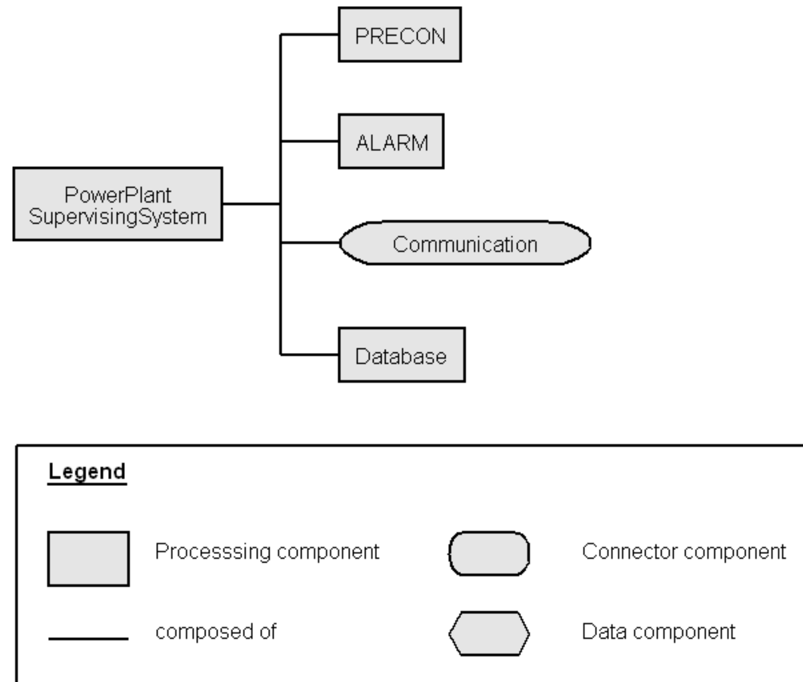

Figure 2.17: component refinement tree resulting from step 1

ized communication infrastructure confirms this choice. `DataBase` was finally introduced to complete this refinement as the need for a database was clearly identified from the requirements. The component refinement tree obtained is presented in Figure 2.17. The boxes represent components, be they processing, data, or connectors, and the lines are composition links.

Step 2: Identification of potential subcomponents from objects specification

Once the basic architecture in place, potential sub components of the basic architecture are obtained from the objects in KAOS specification. Data, processing and connector components are derived to implement `PRECON`, `ALARM`, `DataBase` and Communication components. If in the third step no constraint is assigned to these components, they won't be part of the system prescription.

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

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** ...

One can note that the `Database` component defined in step 1 would have been derived automatically since a database object is present in the KAOS object model.

Since all the components derived from KAOS specifications are data, various processing components need to be defined at this stage to effectively implement the component from step 1. In the same way connector components are specified to permit communication between the newly defined components. The latest two components forementioned are an example. At the next stage it will be decided which of these components would be part of the final prescription based on the assignment of the goals responsibility. The decision will be guided by the goal diagram. The components of step 1 come from goals present in the goal diagram and are thus roots of goals subtrees. Since goal definitions are expressed in terms of objects one looks, in which subtree is the object from which a component is derived. Of course some objects can be present in the definitions of goals in two different subtrees. The choice is then based on what goal this object is the most related to. This is fairly qualitative though.

Step 3: Selection among potential components via constraints assignment

In this step it is determined which of the subgoals are achieved by the system and they are assigned to the previously defined components. With the goal refinement tree as reference, it is decided which of the potential components of step two would take the responsibility of the various goals. Note that this is a design decision made by the architect based on the way he or she chooses to realize the system. The components with no constraints are discarded, and it is ended 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 it was chosen to keep `FaultInformation`. This choice was motivated by the fact `FaultInformation` is the software representation of `Fault`. Different architects may use different approaches.

It is interesting to note that in the 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. During the first iteration it was assumed that `Communication` component could handle these varying types of requirements. However it was then realized that creating sub components for `Communication` component could help to illustrate these differences. Therefore the sub components - `UpdateDBConnect`, `FaultDetectionEngineAlarmManagerConnect` and `QueryDBConnect` were created. As their name suggests, each of them is responsible for the communication in different parts of the system. It was therefore easier to illustrate the different time and security constraints needed for each of them. Note that the fault tolerance requirement introduced after obstacle analysis constrains the connector linking the processing components in charge of fault detection and alarm management, namely `FaultDetectionEngine` and `AlarmManager`.

Please find below 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 TimeConstraint = 1 s
Secure
Maintain[FaultTolerantCommunication(PRECON,ALARM)]

Composed of /

Uses /

Figure 2.18 shows the resulting component refinement tree.

Step 4: Architecture refinement to achieve non-problem goals

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.

During step 2, the fault-tolerant constraint has been added on the connector linking `ALARM` and `PRECON`. However the architecture in itself was not constrained so to ensure fault tolerance. Additional architectural constraints were thereby added to the connector and two copies of both `PRECON` and `ALARM` were introduced. The constraints added state for example that only one copy should work at a time and that whenever a failure of the active copy occurs the other one should take the relay.

A comprehensive list of additional constraints together with the modified system prescriptions is presented in Appendix B.2.

Box-and-line diagram

Once the architecture created, a box-and-line diagram was added to illustrate the various components and connectors. The component tree created as a result of the three steps did not show how the various components are

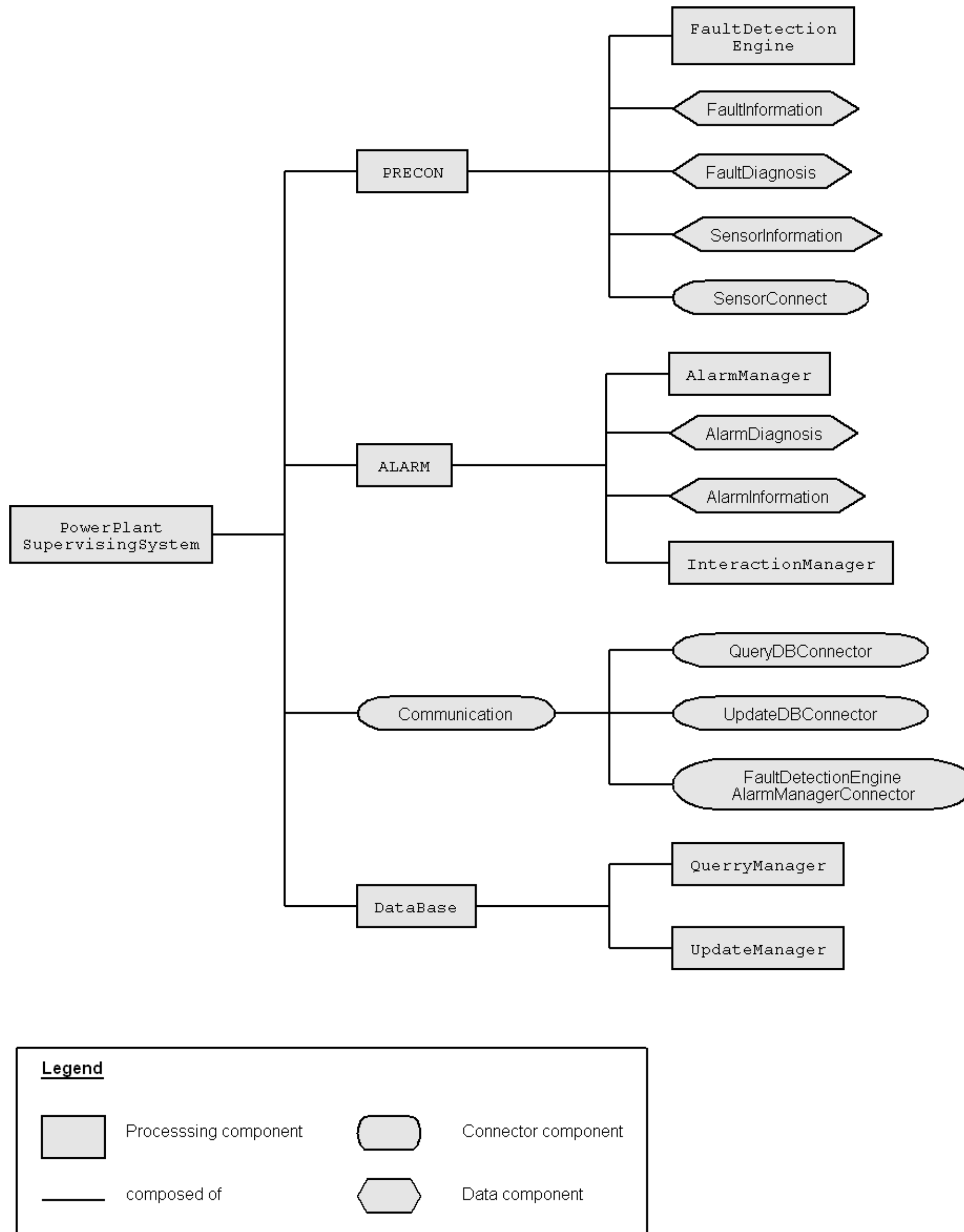

Figure 2.18: Component refinement tree resulting from step 3

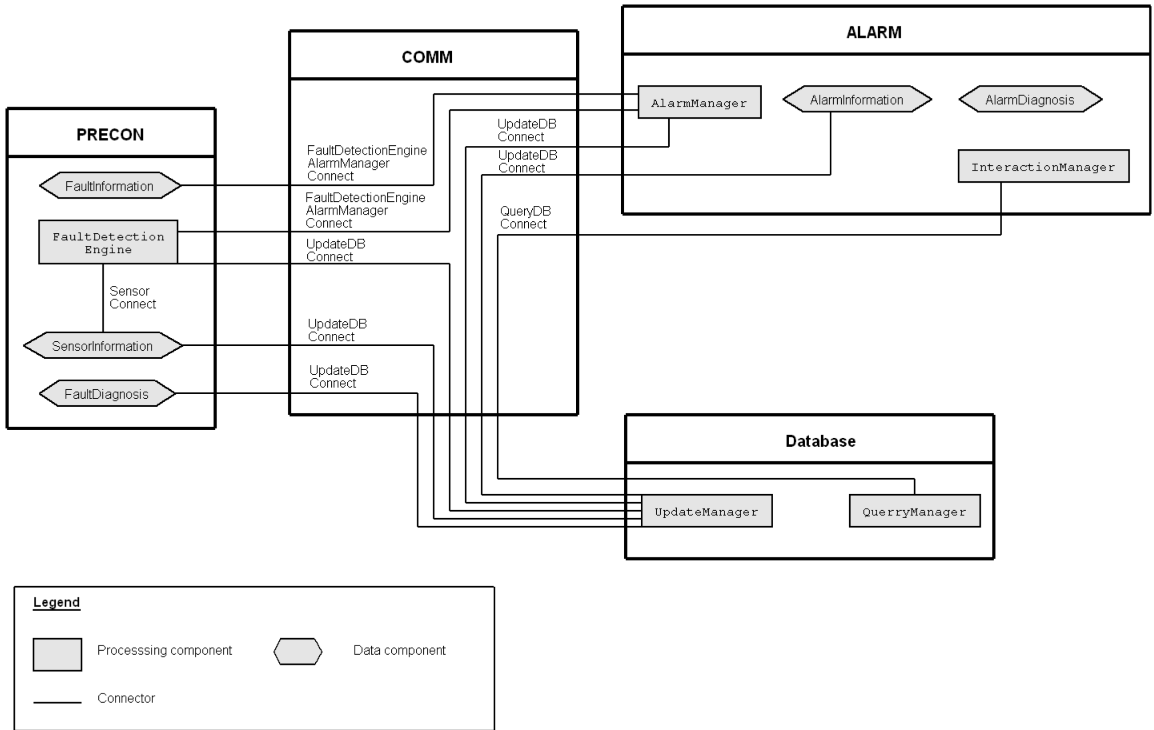

Figure 2.19: Box-and-line diagram

linked through the connectors. The box-and-line diagram helps in visualizing this and gives a more complete view of the architecture. Figure 2.19 shows the resulting diagram.

2.3.3 Comparing the Resulting Architectures

This section will focus on derived architecture characteristics strictly speaking. The general differences such as the formalism used, the level of details provided, the way non-functional requirements are addressed will be examined in Section 2.4.3. First the architectures will be examined with regard to their components in order to check whether they share a subset of components. If some components are missing in an architecture, the reason will be stated. Secondly the presence of an internal structure will be discussed. The ability of the architecture to meet the architectural constraint will then be assessed. Finally the way the two architectures deal with the non-functional goals imposing fault-tolerant communication between PRECON

and ALARM and ensuring accurate data between Acquisition Unit and IMS will be inspected.

For reporting purpose, it will thereafter be referred to the architecture derived by the KAOS method by architecture number one and to the one derived using the Preskriptor process by architecture number two.

The two derived architectures have similarities but also differences. First a set of components are common to both architectures. Among those the two main components of the system, namely PRECON and ALARM, the communication facilities and the database. The functions performed by those components are roughly equivalent except for PRECON. In fact, the first architecture assigns to it as sole function fault detection while the second one adds it sensors management. This results into two additional components in the first architecture, one for data acquisition and the other for data validation and sensor operation checking. Since fault detection and sensor management are two separate concerns, the first architecture can be considered as better with respect to the *high cohesion* design heuristic.

An other important aspect is that the second architecture is hierarchically structured while the first one is not. The components are considered as a whole in the first architecture while they have an internal structure in the second one. The case of PRECON is symptomatic. It is composed of four subcomponents – one processing, two data, and one connector component.

One architectural constraint identified from the domain was the use of a centralized communication infrastructure. In both architectures this requirement is met. This is clearly the case in the first one. One can convince himself by a single at Figure 2.16. This results from the application of centralized communication style defined in Figure 2.12. The situation is less obvious in the second architecture because of the multiple connectors defined. Nonetheless all the connectors used for interaction between PRECON, ALARM and Database belong to the same component, namely Communication. The communication can thereby be said centralized. However, there is some internal communication into the PRECON component. It results from the fact that FaultDetectionEngine needs the measured values of the variables monitored by sensors stored in the SensorInformation component in order to detect faults in the power plant.

The obstacle analysis has enabled to identify the fault tolerance requirement for the communication between PRECON and ALARM. The way both architectures address this issue have common points. Two copies of the components are introduced. There are nevertheless some differences. In the first architecture, a connector links the copies and each copy of PRECON is linked to one copy of ALARM. In the second architecture, there is no linking

Criterion	Architecture 1 (KAOS)	Architecture 2 (Preskriptor)
Components	PRECON ALARM COMM DB Acquisition Unit IMS Consistency Maintainer	PRECON ALARM Communication DB
Hierarchical Structure	no	yes
Centralized communication	yes (COMM)	yes (Communication)
Fault tolerance	yes	yes
Consistency maintainer	yes	not needed

Table 2.1: Comparison between the resulting architectures

connector between the copies and only one connector is used to manage interactions between the two copies of `PRECON` from one side and the two copies of `ALARM` from the other side. Moreover the semantic of the different connectors is not very clear in the first architecture. Thanks to the additional constraint introduced the situation is somewhat better in the second case.

The last difference arises from the fact that in the second architecture, the acquisition of data is grouped together with data validation and sanity checkings for sensors. As they are performed in the same component – `SensorInformation` – there is no need for a consistency maintainer like the one present in the first architecture.

All these considerations are summarized in Table 2.1.

2.4 Discussion

2.4.1 Evaluating the Methods

This section will evaluate the two methods used. In a first time the derivation process will be examined. Since in both cases it consists of intermediate steps, each one will be carefully examined. Various concerns will be reviewed such as the easiness of application, the level of guidance/freedom provided/allowed, the issues encountered during the architecture derivation. In a second time the formalism used will be studied considering its advantages and disadvantages in terms of capabilities/limitations . Thirdly the mapping from the requirements will be examined. Finally the remaining gap between the final architecture and the implementation will be discussed.

The KAOS Method

Once the requirements finalized, the first step allows to get an abstract dataflow architecture. Dataflow architecture is obtained by using functional goals assigned to software agents. The agents become architectural components and then dataflow connectors are derived from data dependencies. This stage is fairly systematic and its application is thus straightforward. The abstract dataflow architecture can be obtained from the agent context diagram by suppressing all the environment agents. The guidance is here maximal and there is no doubt that this step can be automated easily. An important issue here is that the dataflow connectors do not reflect perfectly the actual data flows but the data dependencies. And those two are not always equivalent. Moreover, "absorbent" components appear when some agent do not control any variables. It was the case for `COMM` and `DB` since, due to their inherent function – communication and storage, it could not be said they control the variables transmitted/stored.

In the next stage architectural styles are applied to meet architectural constraints. This step is the most qualitative. Only one style is mentioned in [27] and it does not match the architectural constraints of the system (centralized communication). A new transformation rule had to be designed. The design of the transformation rule was driven by two facts:

1. *initial situation*
   `COMM` was one of the absorbent components discussed here above.

2. *desired situation*
   The architecture needs to reflect the actual data flows by having all the communications centralized through the `COMM` component.

This enterprise was almost a complete success. The only remaining problem is that DB is still an absorbent component although the information on sensors are for example provided by the database to PRECON. The architectural constraint was nonetheless achieved since all the communications are centralized through the COMM component.

The third step prescribes the use of patterns to achieve non-functional requirements. Various sample patterns are given in [27], however they are only suggested. Their description is very partial and informal, leaving lots of shadow areas. First their applicability condition is very vague, defined only by an initial configuration of components in terms of box and lines and by the name of non-functional goal concerning the initial architecture fragment. Secondly the resulting refined fragment of architecture often leads to the introduction of new connectors and components. Neither the components nor the connectors introduced are described in terms of operations performed or data carried. For example, fault-tolerant communication was introduced between PRECON and ALARM thanks to the pattern presented in Figure 2.14 but no information is provided about how the two copies communicate, how they are coordinated and how the surrounding components are affected. Similarly, for the consistency maintainer pattern (see Figure 2.15) applied on Acquisition Unit and IMS, the consistency maintainer component has no associate operations and there is no semantic for the newly introduced connectors. Those investigations lead to the discovery of an other issue: combination of patterns. When two different patterns can be applied on the same components the method does not provide any guidelines. Yet, this situation brings many questions:

- Do the patterns conserve their efficiency if they are combined?
- If not, how to choose the one to apply?
- Does the order of pattern application matter?
- Once one pattern has been applied, will the resulting piece of architecture still match the applicability condition of the other?

Finally, some non-functional requirements are not handled by the proposed patterns. For example the performance requirements so important in real-time systems are not ensured by any patterns.

The KAOS method uses a graphical formalism which is completed by formal definitions coming from the requirements specifications. A structural view of the architecture is provided by a box-and-line diagram, boxes representing components and arrows connectors. This diagram gives a good

overview of the general structure of the system. The components are described by a set of operations that are formally defined in temporal logic. The connectors are specified by the data they carry if they are dataflow while no precise notation is introduced otherwise. The description of the connectors is so far not sufficient. The way the different components interact needs to be described in a precise way. In fact the behavioral view of the architecture is globally missing. Operations describe pieces of behavior of components but there is until now little information on how this different operations form the global behavior of the components. Moreover there is no information on how the different components coordinate their effort to achieve the general system behavior.

The mapping from requirements to architecture is first based on agents. Each agent ensuring a functional goal becomes a software component. The operations ensuring the functional goals are part of its description. The controlled/monitored variables – described in the object model – are used to derive dataflow connectors. Patterns are used to achieve the non-functional requirements. Each model is used and it can thereby be said that the KAOS method uses fully the requirements specifications.

As explained in Section 1.2, architecture should serve as a basis for the design. This method provides a well structured architecture whose functions of the different components are clear thanks to operations specifications. The remaining work concerns mostly the description of the behavior of the components, the way they interact through connectors and how they coordinate their effort to act as a whole. The design phase comes thereafter, specifying the needed data structures, algorithms, communications protocols and so on. The method provides so a good structural basis for the design but is unsatisfactory with respect to the description of the behavior of the system.

The Preskriptor Process

In the first step the basic prescription is derived from the root goal of the system and the knowledge of the other systems it has to interact with. It was difficult to determine how to start and how much to try to do in the first step. The dilemma was the following: what decisions regarding the architecture are made at step one? Is a root component simply assigned or is it needed to anticipate the next steps and to have a basic thought-out structure? The second solution was chosen by defining four subcomponents to the root one, namely `PRECON`, `ALARM`, `Communication` and `Database` because a general structure was thought to be needed.

In the second step objects in the KAOS specification are used to derive potential sub components of the basic architecture. This step was very systematic and did not present any major difficulties. Nonetheless the possibility of using agents instead of objects to derive components was discussed. In fact, it was initially thought at using the `IMS` agent as a sub component but `SensorInformation` (which is an object) was finally used instead.

In the third step an appropriate degree of refinement of the goal refinement tree is selected. At this point the sub goals achieved by the system are assigned to the sub components created in step two. The component that are not constrained by any goals are discarded. The decision of the assignment of goal to a particular component is left to the architect. The method does not provide any guidance. It was sometimes difficult to choose to which component assign a goal. The eventuality of sharing the responsibility of a goal between two components even came up.

The fourth step aims at satisfying the non-problem requirements. The goal of this step is clear but there is no real methodology to achieve this. Once again this lack of guidance was an obstacle to overcome. In order to describe more precisely the architectural implications of the fault tolerance requirement, it was decided to add multiple copies of `PRECON` and `ALARM` similarly to what has been done with the KAOS method and to add a few constraints to the connector linking. All these were personal decisions not motivated by the method.

Architectural prescriptions form the core of the architectural description. They define the components in a hierarchical way. Each component is described by its type, the constraints it has to ensure, the various components it is composed of and the components with whom some interaction is performed as well as the connectors used for those interactions. An important characteristic is here that connectors can have associate constraints like any other component. They can therefore be more active than if their only purpose was to carry data or to call a remote procedure. However connectors cannot specify the data passed through them. Two graphical views are used to support prescriptions. The component refinement tree shows the hierarchy of the components while the box-and-line diagram exhibits the actual structure of the architecture. The behavioral view of the system is not addressed. The views proposed are only static.

The Preskriptor process takes as starting point the objects present in the requirements specification. The various goals of the system are then assigned as constraints to those components. Neither the agents nor the operations are used in the derivation process. The Preskriptor process does thus not make a full use of the requirements specifications.

As the components are only described by constraints, nothing is said about how those constraints are actually realized. This is up to the designer. The whole behavior of each component has to be defined. Even the structure is not totally clear since this method focuses essentially on component hierarchy. There is so a lot of remaining work to achieve even before being able to effectively design the system, in terms of implementation choices for example.

2.4.2 Opportunities for Improvements

The KAOS Method

Suggested improvements belong to two main categories: the first dealing with patterns and style description, the second adding a behavioral view to the architecture descriptions.

In Section 2.4.1, patterns description has been identified as an obstacle to the derivation process. They should therefore be better documented. First their applicability condition should be clearly stated presenting the non-functional goal achieved and the conditions the concerned part of architecture has to satisfy. Secondly the resulting piece of architecture must be clearly defined. Introduced components should be described by a set of operations (like any other component) and new connectors by the data carried and by the interaction protocol used. These considerations also apply to style except that architectural constraints replace non-functional goals.

The behavioral view of the architecture is essentially missing from the KAOS method. Pieces of behavior component are described by operations but the way these operations are coordinated to form the global component behavior, the way the different components interact through connectors and the way all the components coordinate their efforts to achieve a global behavior are not addressed. Therefore, a better description of connectors is essential since they provide the mean of communication and coordination between components. Moreover, the way the different operations of a component form its global behavior in terms of a sequential or parallel execution order for example needs to be further detailed.

The Preskriptor Process

The lack of guidance especially in the first step was really felt as a hard obstacle to overcome and therefore the first improvement suggested concerns a better definition of the method. For the various steps, there should be at least explanations of the different possible strategies, their impact on

the resulting prescriptions, their associate advantages and disadvantages. For example in step 1, the issue of the refinement of the root component should be addressed. An other example is the assignment of constraints to components in step 3. Different strategies lead to different prescriptions and the impact of such strategies should be examined.

An other important improvement should be to develop further the method concerning the satisfaction of non-functional requirements. The way they are handled at the present time is all but clear. There is no methodology to achieve them and very few examples are addressed in [8, 9, 10]. The method provides so far too little support to be useful.

2.4.3 Comparing the Methods

The first important thing to note is that both architectures ensure the satisfaction of all functional requirements, by different means, though. The KAOS method takes every functional goal and creates a component from the responsible agent. The derived component is described by the operations operationalizing the goals so ensuring its satisfaction. The Preskriptor process derives components from objects and uses the goals to constraint them. The KAOS method ensures thus goals satisfaction by adding needed operations to components while the Preskriptor process constraints the components.

The most significant difference is that the KAOS method is more "low level". The components are described together with the operations that they have to perform creating a more rigid design. The Preskriptor process uses an architecture prescription language which tends to be more "high level". This allows the designer more freedom to pick a better solution at a low level. This results in significantly more work for the designer in the later case.

An other important difference is that the KAOS method offers more guidance. Two different architects starting from the same requirements specifications would probably obtain very similar architectures with the KAOS method while the resulting architectures could be very different with the Preskriptor Process.

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 is more hierarchical in nature. An additional box-and-line diagram was needed to better explain the component interactions. However both views though different were useful. And the ability to describe an architecture in terms of hierarchical structures

could also be an improvement to the KAOS method.

In both cases a behavioral view of the architecture was not really present. In the KAOS method, parts of the component behavior are implicitly present through operations definitions while in the Preskriptor process behavior of the system is not treated at all.

The mapping from requirements is also quite different since agents are used in one case and objects in the other one to create components. Moreover the KAOS method uses all the different models of the requirements specifications while the Preskriptor process only uses the goal and the object model.

The way non-functional requirements are addressed is also different. The KAOS methods uses patterns to ensure them while Preskriptor achieves them by further constraining the components.

Chapter 3

Toward More Precise Architecture Derivation

From Section 2.4.2 two areas of improvement for the KAOS method have been suggested in order to both facilitate the architecture derivation process and to obtain a more complete and precise architecture description.

On the one hand, the behavioral view of the architecture is essentially missing and its addition to the KAOS method would complete the structural view already present.

On the other hand, better descriptions of patterns and styles from the point of view of their applicability conditions and of the non-functional goals/architectural constraints they achieved would enable to make the identification of the suitable pattern/style easier so providing more guidance to the architect in its choices. Moreover, current patterns descriptions are weak with respect to fragment of architecture produced by their application. New components and connectors are introduced but no information describes the operations they performed, the way they interact with pre-existing components. Another important concern with patterns is their application cannot modify the ability of the system to achieve its functionalities. They have to achieve some non-functional requirement while keeping all the other functional requirements satisfied. Find a way to check that patterns do not affect the general behavior of the system would be highly beneficial. In conclusion architecture description could be more powerful on the one hand thanks to the addition of a procedure to specify the component behavior as well as their interactions, and on the other hand via a precise description of architecture parts resulting from patterns application.

This chapter aims at making the architecture description more precise

and will therefore focused on two improvement areas: first bring a behavioral view to the architectural descriptions and secondly on describe precisely the structural and behavioral effects of patterns application. Section 1.2 identifies Architectural Description Languages (ADL) as a good mean to describe architectures precisely. Indeed they offer formal notations allowing accurate descriptions of various aspects of the architecture such as its structure, its behavior, the presence of style and formal reasoning capabilities. The use of an ADL will therefore be explored to this end.

The comparative study presented in Section 1.2 serves as a basis for the choice of the ADL. Considering the particular needs of this problem Wright appears as the best solution. Indeed, the main concern is the behavioral description of the architecture and Wright allows the description of both individual component behavior and their interactions through connectors. Moreover, Wright supports styles and allows some formal reasoning capabilities as for example, the possibility to check the absence of deadlocks in the architecture. Another element in favor of Wright is its well-known formalism since it uses a CSP-like notation.

This chapter is structured as follows: Section 3.1 introduces the Wright architecture description language, Section 3.2 presents ways to derive architecture descriptions in Wright, Section 3.3 describes the application of the derivation mechanisms to the power plant supervisory system and Section 3.4 presents a comprehensive description of the two patterns applied to the power plant supervisory system.

3.1 Wright

In order to provide the reader with the necessary basis to understand this chapter, Wright will be presented. For further details please refer to [1]. In a first time its syntax will be exposed. Then will be examined the three architectural abstractions present in Wright, that is, *components*, *connectors* and *configurations*. Next the semantic of components and connectors will be informally explained in order to show how the global behavior of the system is derived from their specifications. Finally the various kinds of possible analysis will be briefly discussed.

The notation used is a subset of CSP, containing the following elements:

- **Processes and Events:** A process describes an entity that can engage in communication events. Events may be primitive or they can have associated data (as in e?x and e!x, representing respectively data

input and output). The simplest process, STOP, is the one that engages in no events. The event $\surd$ is used to represent the "success" event. The symbol § is used to denote a successful terminated process, formally § $\stackrel{def}{=} \surd \rightarrow$ STOP. A distinction is made between initiating an event and observing an event. Practically an event initiated by a process is written with an overbar while an event observed by a process is written without overbar.

- **Prefixing:** A process that engages in event e and then becomes process P is denoted $e \rightarrow P$.

- **Deterministic choice:** $P[]Q$ refers to a process that can behave like P or Q depending of its environment (the environment relates to the other processes interacting with the process) .

- **Non-deterministic choice:** $P \sqcap Q$ denotes a process that can behave like P or Q, the choice being made (non-deterministically) by the process itself.

- **Parallel composition** $P \| Q$ refers to the parallel execution of processes P and Q.

- **Condition:** The **when** operator is used to have a different process behavior depending on some condition on its state variables.
  $P_v = \begin{cases} Q & \textbf{when } A(v) \\ R & \textbf{otherwise} \end{cases}$ denotes a process over variable v that behaves like Q or R depending on the truth value of $A(v)$.

- **Named processes:** Process names can be associated with a process expression through the **where** operator.

In Wright, the description of a **component** has two important parts, the *interface* and the *computation*. An interface consists of a number of ports. Each *port* represents an interaction in which the component may participate and describes its behavior at that particular point of interaction. The computation section explicits what the component actually does. The computation carries out the interactions described by the ports and shows how they are tied together to form a coherent whole. It consists of all the interactions described in the port fields together with internal processing and combines all these information to provide a full specification of the component behavior.

A Wright description of a **connector** divides it into a set of *roles* and the *glue*. Each role specifies the behavior of a single participant in the interaction. The glue specifies how the activities of the different roles are coordinated. In a similar way to the computation field of components, the glue consists of all the interactions described in the role field together with internal processing. It composed these data to specify how the participants work together to create an interaction so providing the full behavioral specification.

In order to get a complete system architecture, the components and connectors of a Wright description must be combined into a **configuration**. A configuration is a collection of component instances combined via connectors. A configuration consists of the definition of *instances* and *attachments*. Instances are necessary since there may be many components or connectors of the same type. Attachments define the system topology by linking components instances via connectors instances. Note also that Wright supports hierarchical descriptions.

The behavior of an architectural configuration is constituted by each behavior of the individual components, each operating independently except the fact they are coordinated by the glue of the connectors to which they are attached. The computation of each component forms a part of the overall behavior, where the order in which the computations occur and the data transfer from one to the other is coordinated by the connectors.

Wright provides also means to analyze the architecture. Classic checks include detection of deadlocks, starvation, races conditions. Another important issue is to check whether attached *ports* and *role* are behaviorally compatible. As Wright uses a subset of the CSP notation, all the analysis are performed using the FDR[13] model checker.

3.2 Deriving Architectures in Wright

3.2.1 Integration within the KAOS method

Two solutions can be imagined to derive Wright specifications from the architecture description resulting from the KAOS method. The first one tightly couples the two derivation processes where Wright specifications are constructed incrementally on the basis of the intermediate architectures produced by the KAOS 3 steps process. The second possibility is to derive Wright specifications only once the whole architectural description has been built.

The first alternative has been chosen for two reasons. First one can convince that it is far more easy to derive Wright specification from the initial abstract dataflow architecture than from the final pattern-based architecture. Assuming the existence of well defined transformation rules for patterns and styles, the continuation of the Wright derivation process should not present any major problems. Secondly incremental construction of Wright specifications enables to validate the architecture after each transformation ensuring that this transformation does not prevent from satisfying any functional requirements.

Of course this solution has the drawback to demand more work. Three Wright specifications are built instead of one. Nevertheless it is believed that the advantages forementioned largely counterbalance this inconvenient. Moreover appropriate tools can support the process so as to minimize the impact of extra work.

As a result from this approach, this subsection will focus on how to derive a Wright specification from the abstract dataflow architecture produced by the first step of the KAOS method while Section 3.4 aims at refining this initial Wright specification by patterns application.

3.2.2 Structure

Even if not the prime objective, architecture structure is an integral part of a Wright description and must therefore be handled. Structure description in Wright consists of the identification of the different components and connectors types as well as the description of their interface (via *ports* and *roles*, respectively), and the way those components and connectors are linked so as to form the system topology (via *instances* and *attachments*). It is important to note that two levels of description exist. Since there may be many *instances* of the same component or connector, the concept of *type* is introduced. The type describes the properties of the component or connector while instances are actual examples of them in use. As the structural view is already present under graphical form in the abstract dataflow architecture and as components and connectors are the main constitutive elements in both formalisms, the mapping to Wright is straightforward. It can be expressed by the following rules. Rules 1 and 2 express the correspondences between components and connectors types in KAOS and Wright. Rules 3 and 4 deal with the instance level. Rule 5 is concerned by the system topology.

Rule 1 *Let C^{KAOS} be a component type in the KAOS architecture descrip-*

tion. Let n be the number of dataflow connectors coming from or to C^{KAOS}. *The translation of* C^{KAOS} *in Wright is a component* C^{Wright} *with n ports* $p_1, \ldots, p_n$.

component C^{Wright}
  port p_1
  $\vdots$ $\vdots$
  port p_n

Rule 2 *The dataflow connector type in the KAOS architecture description is translated in Wright by a connector type Dataflow with two roles, namely Producer and Consumer.*

connector *Dataflow*
  role *Producer*
  role *Consumer*

Rule 3 *Let* C^{KAOS} *be a component type in the KAOS architecture description and* C^{Wright} *its correspondence in Wright. For each instance of* C^{KAOS} *present in the KAOS architecture description an instance* C^{Wright} *is defined in the Wright specifications.*

Rule 4 *For each instance of a dataflow connector in the KAOS architecture description an instance of the Dataflow connector type is defined in the Wright specifications.*

Rule 5 *Let* C_1^{KAOS}, C_2^{KAOS} *two components linked by a dataflow connector* D^{KAOS} *coming from* C_1^{KAOS} *to* C_2^{KAOS} *in the KAOS architecture description. Let* C_1^{Wright}, C_2^{Wright} *be the corresponding Wright components and* $Output$ *and* $Input$ *be the ports used by each one to interact with the other. Let* $Dataflow^{Wright}$ *be the corresponding Wright dataflow connector with its two roles* $Producer$ *and* $Consumer$. *The corresponding instances* c_1^{Wright}, c_2^{Wright} *and* d^{Wright} *of the components and connectors are attached in Wright as follows:*

attachments $c_1^{Wright}.Output$ **as** $d^{Wright}.Producer$
  $c_2^{Wright}.Input$ **as** $d^{Wright}.Consumer$

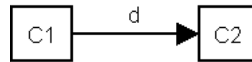

Figure 3.1: Fragment of an abstract dataflow architecture

Considering for example two components $C1$ and $C2$ linked through a dataflow connector d. From the graphical representation of Figure 3.1 and with the help of the rules defined, the following Wright specification is obtained without any difficulty.

configuration Example

  component C1
    port Output
  component C2
    port Input
  connector Dataflow
    role Producer
    role Consumer
  instances a: C1
    b: C2
    d: Dataflow
  attachments a.Output **as** d.Producer
    b.Input **as** d.Consumer

end Example

3.2.3 Behavior

Connectors

Since dataflow connectors are the only type of connectors present in the abstract dataflow architecture, they will be our only concern. A dataflow connector expresses the interaction of a component providing some data – the *producer* – with a component needing this data – the *consumer*. One can consider two alternative behaviors: the "push" and the "pull" behavior. In the push behavior (see Figure 3.2), the producer initiates an event with data attached to it each time new data have been produced. In the pull behavior (see Figure 3.3), each time new data are produced, the producer

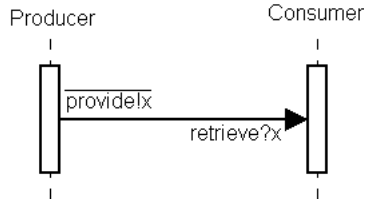

Figure 3.2: Example of interaction scenario for the "push" behavior

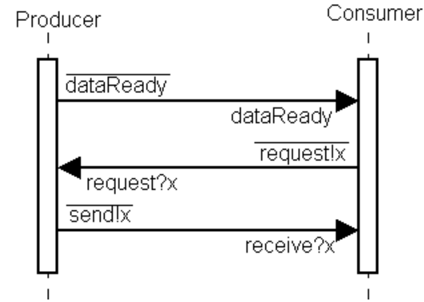

Figure 3.3: Example of interaction scenario for the "pull" behavior

notifies the consumer. It is the consumer's duty to retrieve the data by sending a request to the Producer.

This leads to the two following specifications for dataflow connectors:

connector Push-Dataflow

> **role** Producer = $\overline{provide!x}$ → Producer ⊓ §
>
> **role** Consumer = retrieve?x → Consumer [] §
>
> **glue** = Producer.provide!x → $\overline{Consumer.retrieve?x}$ → **glue** [] §

connector Pull-Dataflow

> **role** Producer = $\overline{dataReady}$ → Producer
> [] request?x → $\overline{send!x}$ → Producer ⊓ §
>
> **role** Consumer = dataReady → $\overline{request!x}$ → receive?x → Consumer
> [] §
>
> **glue** = Producer.dataReady → $\overline{Consumer.dataReady}$
> → Consumer.request!x → $\overline{Producer.request?x}$
> → Producer.send!x → $\overline{Consumer.reveive?x}$ → **glue** [] §

The choice between the "push" and the "pull" behavior depends on the particular case. If the consumer could be sometimes overloaded, perhaps it would be better to chose the "pull" behavior in order to enable it to handle the new data once it is ready. If the producer cannot keep the produced data due to a lack of memory, the "push" behavior is more appropriate.

Components

Parts of components behavior are expressed in the architecture description by the operations a component has to perform. The idea is to use as much as possible this information so as to extract a global component behavior. A set of heuristics will be presented in order to derive the Wright specification of a component given its operations specifications in temporal logic. These heuristics result from an attempt to infer generalizations from concrete examples. In fact, the system was first specified partially in Wright and these heuristics are a tentative to structure the derivation process. One should keep in mind that these are only heuristics and not proven correct rules. They nonetheless provide a useful help in the derivation of Wright specifications.

Heuristics are grouped into two main parts. First it is tried to translate a KAOS operation specification in Wright. Secondly the control flow is inferred so as to combine the different operations to obtain the full behavior specification of the component. The presentation of each heuristic consists of three parts. The first one explains in an intuitive way what is the justification behind the heuristic. The second part defines the heuristic as formally as possible. Finally the third part applies the defined heuristic on examples coming from the power plant supervisory system specification.

Translating a individual operation

Pre-, post- and trigger conditions As explained in Section 1.1.2 operations are defined in KAOS by a set of fields. The input/output fields define the object(s) received in parameters and modified by the operation respectively, so defining the operation signature. Domain pre-/post conditions describe the elementary conditions on input and output states in the domain. Required pre-/post conditions prescribe additional conditions on input and output states that are necessary for ensuring some goals. Required trigger conditions capture sufficient conditions on input state that require the immediate application of the operation.

In Wright two constructs can be used to represent conditions: events, and conditions preceded by the **when** operator. As the when operator is used to describe different behaviors depending on the truth value of some condition, it is only appropriate to express trigger and preconditions. Postconditions are therefore always expressed by events. But what about preconditions? It depends. A precondition of an operation can also be a postcondition of another operation of the same component. In that case, the precondition

will be expressed as an event. If it is not the case, two possibilities remain; either they are postconditions of an operation performed by another component or they are only preconditions of that particular operation. The former case implies some transmission from that other component to the considered component in order to notify the satisfaction of the condition. Otherwise the precondition will be forever false since the component will not know the operation precondition has just become true. Who says transmission says there will be an event coming from one of the ports and the precondition will thus be represented as an event. The latter case will be expressed through the when operator. Classical situations include preconditions representing the state of an internal variable or some property of the environment. The preceding considerations can be summarized by the definitions of the following heuristics.

Heuristic 1 *Let C be a component,* $Op_1, \ldots, Op_n$ *be its operations.*
The postconditions (required or from the domain) of $Op_1, \ldots, Op_n$ *are expressed in Wright by events.*

Heuristic 2 *Let C be a component,* $Op_1, \ldots, Op_n$ *be its operations.*
The pre-(required or from the domain) and trigger conditions of $Op_1, \ldots, Op_n$ *can be expressed either by events or by condition preceded by the* ***when*** *operator. Conditions preceded by* ***when*** *are used when the pre- or trigger conditions refer to the state of an internal variable or some property of the environment. Events are used otherwise.*

These heuristics will now be illustrated with an example. Considering the operation `SwitchSensorOff` whose pre- and postconditions are given by:

DomPre	s.Status = on
DomPost	s.Status= off
ReqTrig	$\neg$ s.WorkingProperly

The postcondition s.Status=Off will be expressed by the event TurnOff(s) as prescribed by heuristic 1. Both the pre- and the trigger conditions refer to the environment since the variable s denotes a sensor and `Sensor` is an environment agent. They will be therefore expressed by the condition s.Status=on $\wedge$ $\neg$ s.WorkingkingProperly as said in heuristic 2.

Limitations of Wright regarding to temporal logic As Wright does not support temporal logic there is an inevitable loss of semantic resulting from the translation of pre-, trigger and postconditions into Wright

constructs. The time constraints will be in particular not translated. Nevertheless it is sometimes possible to be more restrictive than asked by the pre-, trigger and postconditions. Consider the typical example of bounded achieve goals expressed by $A \Rightarrow \Diamond_{\leq ts} B$ that produces a trigger condition of the form $\neg B \mathbf{S_{t-1s}} A \wedge \neg B$. The trigger condition can be enforced to A so transforming the goal in an *immediate* achieve instead of a *bounded* achieve. The operationalization looses in that case its minimality property but keeps its completeness and consistency.

Data transmission Data transmission is a particular case of the two preceding heuristics since Wright has a dedicated construct for data transmission. As explained in Section 3.1 data can be associated to events via $?d$ for a reception and $!d$ for a transmission. In the abstract dataflow architecture, data transmissions are expressed through dataflow connectors. Assume that a dataflow connector links C1 and C2 via ports P1 and P2 and transmits some data d from C1 to C2, there must be an event in the specification of C1 specifying the sending of d and an event in the specification of C2 specifying the reception of d. This is expressed by the following heuristic:

Heuristic 3 *Let C_1, C_2 be two components linked by a dataflow connector via ports p_1 and p_2, let d be the transmitted data from C_1 to C_2. There must be an event e_1 of the form $\overline{p1.e_1!d}$ in the specification of C_1 and an event e_2 of the form $p2.e_2?d$ in the specification of C_2.*

Looking at the abstract dataflow architecture presented in Figure 2.11 one can for example note the dataflow connector between the component `Acquisition Unit` and `Database` carrying sensor information (si). Assuming that $ToDB$ and $FromAcq$ are the ports used by `Acquisition Unit` and `DB` to communicate, there will be an event $\overline{ToDB.transmit!si}$ and an event $FromAcq.receive?si$ in the specifications of `Acquisition Unit` and `DB` respectively.

Operations Once the various pre-, trigger and postconditions have been expressed by events or conditions in Wright, they have to be linked to form the operation. As pre- and trigger conditions precede postconditions, there should be a sequential order between them. The pre- and trigger conditions that are expressed by events are therefore linked to the corresponding postconditions using $\rightarrow$ in Wright. The pre- and trigger conditions expressed by a condition are added through the **when** operator. Note that in some

case all the preconditions are expressed by events or by conditions. Consequently three cases should be distinguished:

1. all the pre- and trigger conditions are expressed by events
2. all the pre- and trigger conditions are expressed by conditions
3. the pre- and trigger conditions are expressed by events and conditions

Depending on the case the operation specification can be constructed as stated by the following heuristic:

Heuristic 4 *Let Op be an operation. Let pre be the event expressing pre- and trigger conditions if there exists. Let cond be the condition expressing preconditions if there exists. Let post be the event expressing postconditions. Depending on the case the Wright specification of the process representing Op is given by:*

1. *if all the pre- and trigger conditions are expressed by events*
   $pre \rightarrow post$
2. *if all the pre- and trigger conditions are expressed by conditions*
   post **when** *cond*
3. *if the pre- and trigger conditions are expressed by events and conditions*
   $pre \rightarrow post$ **when** *cond*

Wright allows to associate a process name with a process expression. A process in Wright is an entity that can engage in events. Since at least the postconditions of the operations are expressed by events, operations are thus processes. In order to make the mapping from the architecture description resulting from the KAOS method as clear as possible processes corresponding to operations should be associated with the name of the operations.

Heuristic 5 *Let Op be an operation.*
The process representation of Op in Wright should be associated with the name of Op.

Continuing with the example of `SwitchSensorOff`, its complete Wright specification can be derived using heuristic 4 and the resulting process can be named SwitchSensorOff as advised by heuristic 5.

$$\text{SwitchSensorOff} = \text{TurnOff(s)} \quad \textbf{when} \quad \begin{array}{l} \text{s.Status=on} \\ \wedge \neg\ \text{s.WorkingkingProperly} \end{array}$$

Infering the control flow

Sequential Composition First in order to facilitate the following explanations two definitions are introduced: the global precondition and global postcondition. They formalize the fact that in order to perform an operation, all its preconditions, be they required or from the domain, and at least one of its trigger condition must hold. Similarly, once an operation has been performed, all its postconditions, be they required or from the domain, must hold. The following definitions result from those considerations.

Definition 1 *Let Op be an operation,* $DomPre_1, \ldots, DomPre_m$ *be its associate domain preconditions,* $ReqPre_1, \ldots, ReqPre_n$ *be its associate required preconditions and let* $ReqTrig_1, \ldots, ReqTrig_p$ *be its associate trigger conditions,* $DomPost_1, \ldots, DomPost_q$ *be its associate domain postconditions,* $ReqPost_1, \ldots, ReqPost_r$ *be its associate required postconditions.*

$$\begin{array}{lcl} DomPre(Op) & = & \bigwedge_{i=1}^{m} DomPre_i \\ ReqPre(Op) & = & \bigwedge_{i=1}^{n} ReqPre_i \\ Trig(Op) & = & \bigvee_{i=1}^{p} ReqTrig_i \\ DomPost(Op) & = & \bigwedge_{i=1}^{q} DomPost_i \\ ReqPost(Op) & = & \bigwedge_{i=1}^{r} ReqPost_i \\ Pre(Op) & = & DomPre(Op) \wedge ReqPre(Op) \wedge Trig(Op) \\ Post(Op) & = & DomPost(Op) \wedge ReqPost(Op) \end{array}$$

It will now be attempted to characterize the concept of sequentiality between two operations. Let C be a component, and Op_1 and Op_2 two of its operations. The question is: "Is there any relationship between $Post(Op_1)$ and $Pre(Op_2)$ that constraints Op_1 to be applied before Op_2?". By definition of $Post(Op_1)$ and $Pre(Op_2)$, $Post(Op_1)$ holds after the application of Op_1 and $Pre(Op_2)$ must hold in order to apply Op_2. Intuitively $Post(Op_1) \supset Pre(Op2)$[1] appears directly as a condition since this formula states that after applying Op_1, $Pre(Op_2)$ holds since $Post(Op_1)$ implies $Pre(Op_2)$ and Op_2 can therefore be applied. However this is not the sole case where two operations must be applied sequentially. Consider the following example:

$$\begin{array}{l} Post(Op_1) : x = 5 \\ Pre(Op_2) : x = 5 \wedge y = 6 \end{array}$$

[1] $\supset$ will be used instead of the classical graphic notation for implication $\rightarrow$ to prevent any confusion with the Wright operator

Case	Post(Op$_1$)	Pre(Op$_2$)	$\neg Post(Op_1)$ $\vee Pre(Op_2)$	$\neg Pre(Op_2)$ $\vee Post(Op_1)$
1	$x = 5$	$x = 5$	true	true
2	$x = 5 \wedge y = 6$	$x = 5$	true	satisfiable
3	$x = 5$	$x = 5 \wedge y = 6$	satisfiable	true
4	$x = 5 \vee x = 6$	$x = 5$	satisfiable	true
5	$x = 5$	$x = 5 \vee x = 6$	true	satisfiable
6	$x = 5 \wedge y = 6$	$x = 5 \wedge z = 7$	satisfiable	satisfiable
7	$x = 5 \wedge y = 6$	$x = 5 \vee z = 7$	true	satisfiable
8	$x = 5 \vee y = 6$	$x = 5 \wedge z = 7$	satisfiable	true
9	$x = 5 \vee y = 6$	$x = 5 \vee z = 7$	satisfiable	satisfiable
10	$x = 5$	$y = 5$	satisfiable	satisfiable

Table 3.1: Cases where Op_1 and Op_2 must be applied sequentially

In this case, $Post(Op_1)$ is a *necessary* condition for the application of Op_2 even if it is not a *sufficient* condition. Formally $Pre(Op_2) \supset Post(Op_1)$. Those two implications can be rewritten using disjunctions:

$$(1) \quad Post(Op_1) \supset Pre(Op2) \equiv \neg Post(Op_1) \vee Pre(Op2_)$$
$$(2) \quad Pre(Op_2) \supset Post(Op_1) \equiv \neg Pre(Op_2) \vee Post(Op_1)$$

The situation is far more complex than just those two cases. Other situations exist where the fact that Op_1 must be applied before Op_2 is arguable. A non exhaustive set of interesting situations is summarized in Table 3.1.

, Case 1 is the simplest case where $Post(Op_1)$ is equivalent to $Pre(Op_2)$. In cases 2, 5, and 7 $Post(Op_1)$ implies $Pre(Op_2)$. $Post(Op_1)$ is a *sufficient* condition to the application of Op_2. In cases 3, 4, and 8, $Pre(Op_2)$ implies $Post(Op_1)$. $Post(Op_1)$ is a *necessary* condition to the application of Op_2. One can note that for cases 4 and 8 it is more debatable to state a sequential order because it is not sure that $Post(Op_1)$ will contribute to $Pre(Op_2)$. This is due to the presence of the disjunction in the postconditions. One can convince oneself that such a situation is unlikely to happen since it would mean the operation has two different behaviors. Case 6 is also interesting as $Post(Op_1)$ contributes to $Pre(Op_2)$ but without any implication in any direction. Case 10 illustrates a problem that can arise because of different names. If in Op_1 the integer is called x and in Op_2 is called y. The implications will not work even though $Post(Op_1)$ is equivalent to $Pre(Op_2)$ if x is renamed in y. This discussion leads to the definition of the following heuristic.

Heuristic 6 *Let C be a component, Op_1, Op_2 be two of its operations. Assuming that $Post(Op_1)$ and $Pre(Op_2)$ are not equal to the constant true or false. Op_1 application precedes Op_2 application if there exists a renaming of variables such as*

$Post(Op_1) \supset Pre(Op_2)$ *or* $Pre(Op_2) \supset Post(Op_1)$

Two things should be kept in mind: first the heuristic does not say that application of Op_2 takes place straight after Op_1 application but only that Op_2 follows Op_1 application without saying anything about when exactly Op_2 is applied; secondly other operations can have to be performed sequentially; the heuristic is not exhaustive.

To illustrate this heuristic, the operation $SwitchSensorOff$ already discussed and its dual operation $SwitchSensorOn$ will be examined. Their pre-, trigger and postconditions are given by:

Operation	`SwitchSensorOff`	**Operation**	`SwitchSensorOn`
DomPre	s.Status = on	**DomPre**	s.Status = off
DomPost	s.Status= off	**DomPost**	s.Status= on
ReqTrig	¬ s.WorkingProperly	**ReqPre**	s.WorkingProperly

One can note that Pre(SwitchSensorOn) ⊃ Post(SwitchSensorOff) and that Pre(SwitchSensorOff) ⊃ Post(SwitchSensorOn). Heuristic 6 therefore prescribes that there is a sequential order between the two operations wich is logical since to turn a sensor off it has to be turned on before and vice versa. One can also note that although there is a sequential order, the two operations do not have to be successive. Indeed other operations can slot in their application.

Direct sequential composition The next heuristic goes further imposing a direct succession between two operations. The heuristic bases on the milestone pattern for goal refinement. It prescribes that some milestone states are mandatory in order to reach the final one, that is, it imposes that the milestone state precedes the final state. The intermediate goals this pattern produces are `Achieve` goals. These can be operationalized by the bounded achieve pattern. The two resulting operations must therefore be executed sequentially. The two patterns are presented in Figure 3.4.

One can note interesting characteristics. $DomPost(Op1)$ is a necessary condition to $Trig(Op_2)$, that is, $Trig(Op_2) \supset \blacklozenge DomPost(Op1)^2$. It is thus

[2] the ♦ operator is needed because the $A\mathbf{S}B$ operator implies that A has been true but

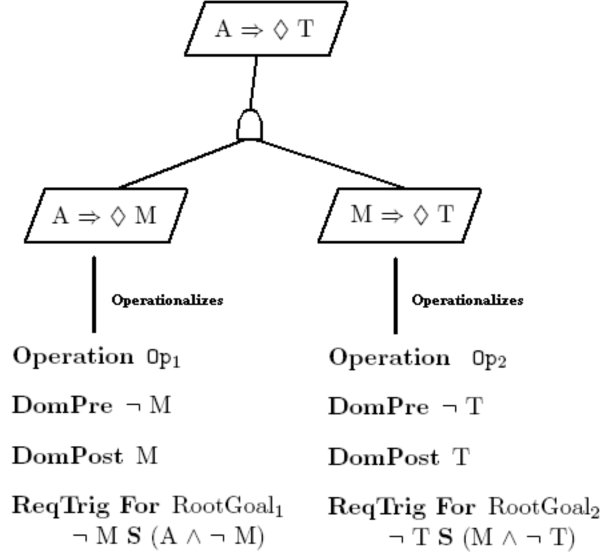

Figure 3.4: Milestone refinement pattern with corresponding operationalizations

logical that Op_1 and Op_2 are applied successively since the completion of Op_1 triggers the application of Op_2 provided the domain precondition holds.

Moreover, if $Post(Op_1) \supset Trig(Op_2)$, then $Post(Op_1)$ triggers immediately the application of Op_2. All this results in the following heuristic:

Heuristic 7 *Let C be a component, Op_1, Op_2 be two of its operations, $ReqTrig_1$, ..., $ReqTrig_m$ be the trigger conditions of Op_2, $DomPost_1$, ..., $DomPost_n$ be the domain postconditions of Op_1, $ReqPost_1$, ..., $ReqPost_p$ be the required postconditions of Op_1. Assuming that none of the conditions is equal to the constant true or false, Op_1 application is directly followed by the application of Op_2 if a renaming of variables exists such as*

$$\exists i \in [1, m], j \in [1, n], k \in [1, p]$$
$$Post(Op_1) \supset ReqTrig_i \bigvee ReqTrig_i \supset \blacklozenge DomPost_j$$
$$\bigvee ReqTrig_i \supset \blacklozenge ReqPost_k$$

The two operations are therefore composed using the → operator in Wright.

---

not that is still true

As an example of the application of this heuristic, consider the two operations `RaiseAlarm` and `SwitchAlarmStatusOn` performed by `ALARM`. Their simplified specification is given by:

Operation	`RaiseAlarm`
DomPre	¬ Raise(fi,a)
DomPost	Raise(fi,a)
ReqTrig	@ Transmitted(fi,PRECON,ALARM)

Operation	`SwitchAlarmStatusOn`
DomPre	powerPlant.AlarmStatus = off
DomPost	powerPlant.AlarmStatus= on
ReqTrig	Raise(fi,a)

As DomPost(RaiseAlarm) $\supset$ ReqTrig(SwitchAlarmStatusOn) the two operations must be applied successively as prescribed by heuristic 7.

Parallel composition The preceding heuristic defines a mean to link two successive operations but not how to compose the others, be they sequentially related or not. The remaining operators to compose operations in Wright are $\sqcap$, [], $\|$. Different choices will result in different behaviors. However one must pay attention to certain important considerations. When two operations are composed using the $\|$ operator, they are executed in parallel and parallelism can lead to *race conditions*. Race conditions occur when two operations try to update the same data at the same time. Although Wright enables their detection, it might be better to ensure their absence by construction. That is why the use of the parallel composition operator between two operations is advocated only provided their **Output** field is different. Moreover operations that are linked sequentially, be they successive or not, cannot be executed in parallel. This leads to the following definition and heuristic.

Definition 2 *Let Op be an operation,* $Obj_1, \ldots, Obj_n$ *the objects present in its output field., the set of objects updated by Op is defined by:*
Update(Op)=$\{Obj_1, \ldots, Obj_n\}$

Heuristic 8 *Let* Op_1 *and* $0p_2$ *be two operations such as* Op_1 *must not precede* Op_2 *and* Op_2 *must not precede* Op_1*.* Op_1 *and* Op_2 *can be composed using the parallel composition operator (*$\|$*) in Wright only if:*
Update(Op_1*)* $\cap$ *Update(*$0p_2$*)* $= \emptyset$

Variable application order For the remaining operations, the choice between [] and $\sqcap$ is essentially guided by the semantic associated to those operators. If at least one of their preconditions refers to the environment, for example because the transmission of some data from another component is needed, then the [] operator is used. If no precondition refers to the environment, then the $\sqcap$. This is expressed by the following heuristic:

Heuristic 9 *Let Op_1 and $0p_2$ be two operations such as Op_1 and Op_2 are not successive operations and cannot be executed in parallel. Op_1 is composed with Op_2 using the [] operator in Wright if at least one of their preconditions refers to the environment, otherwise the $\sqcap$ operator is used.*

Let's come back to the two operations `SwitchSensorOff` and `Switch-SensorOn`. It has been discussed that they are linked by a sequential order but no operator linking them has been imposed so far. Since their precondition typically refer to the environment through Sensor, the [] operator will be used to compose them. The specification resulting from their composition is given by:

SwitchSensorOff [] SwitchSensorOn **where**
SwitchSensorOff = TurnOff(s) **when** s.Status=on
$\wedge$ $\neg$ s.WorkingkingProperly
SwitchSensorOn = TurnOn(s) **when** s.Status=off
$\wedge$ s.WorkingkingProperly

3.2.4 Elaboration of Scenarios

An important by-product of the Wright description is the ability to generate scenarios. An animator for CSP (ProBE) can be used to visualize the behavior of the architectural Wright specification. Of course from a single Wright specification a lot of scenarios can be generated due to the presence of the different components running in parallel. Even within a component, the behavior can lead to many different scenarios because of the presence of a choice operator ($\sqcap$ or []) for example. The scenarios so elaborated can be compared with the ones envisioned during requirements analysis to see if they match. This could be used to validate the architecture with respect to its functional requirements and to detect faults in the translation to Wright. Depending on the case the requirements or the architecture could be corrected. This outlines that the different phases of the software development process are interwoven and that this process is iterative rather than sequential.

3.3 Application to the Power Plant System

This section aims at illustrating the derivation from the abstract dataflow architecture to a Wright description using the various techniques presented in Section 3.2. The Wright specification will be elaborated step by step in order to highlight how the different heuristics can be applied. For briefness purpose, only the fragment of the architecture dealing with `PRECON` and `ALARM` will be examined as presented in Figure 3.5.

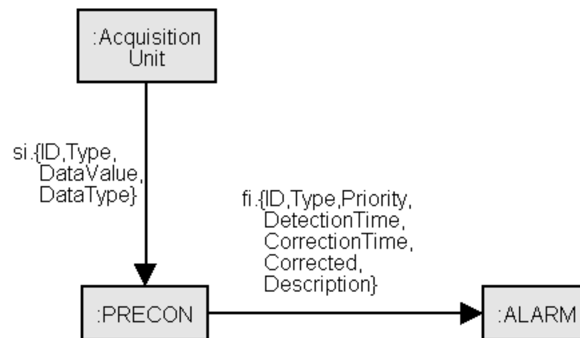

Figure 3.5: Fragment of the dataflow architecture containing `PRECON` and `ALARM`

From the graphical representation a first draft of the Wright specification can be very easily obtained according to what has been explained in Section 3.2.2. The resulting Wright specification is:

configuration PRECON-ALARM

component PRECON
port ALARMOutput
component ALARM
port PRECONInput
connector Dataflow
role Producer
role Consumer
instances Precon: PRECON
Alarm: ALARM
Precon2Alarm: Dataflow

Attachements Precon.ALARMOutput **as** Precon2Alarm.Producer
ALARM.PRECONInput **as** Precon2Alarm.Consumer

end PRECON-ALARM

Once the basic structure in place, the type of dataflow connector can be chosen. Since `ALARM` can be busy with the management of an alarm, the pull-dataflow is selected. So doing `ALARM` will only ask for information on fault when it will be ready to handle them. The specification of the pull-dataflow connector is given in Section 3.2.3.

The specification of the `PRECON` and `ALARM` behavior can now be derived. `PRECON` will be considered first. It has to perform the following operations:

1. Calculate
2. DetectFault
3. SwitchFaultStatusOn
4. SwitchFaultSTatusOff

Consider the first operation: `Calculate`. Its KAOS specification is given by:

Operation `Calculate`

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

Heuristic 1 states that postconditions are expressed through events. The event *CalculationDone* is so created. Figure 3.5 shows that there is a dataflow connector between `Acquisition Unit` and `PRECON` carrying sensor information. Heuristic 3 states there must be an event e of the form $e?si$ in `PRECON` Wright specification. Moreover heuristic 2 states that the

trigger conditions can be expressed by events. The required trigger condition is thereby expressed by the event *AcquisitionInput.receive?si*. One should note two things: first the formalization of the operation states the information on sensors have to be transmitted from `DB` to `PRECON` and not from `Acquisition Unit`, secondly the event does not exactly express all the trigger conditions.

This first problem has already been discussed in Section 2.3.1. It is because the dataflow architecture derived does not always reflect the actual dataflow. The problem is therefore not from the heuristics but from the abstract dataflow architecture itself. For this reason it will next be assumed that sensor information are transmitted from `Acquisition unit`.

The second problem arises from the fact that Wright does not support temporal constraint specification. Part of the information is consequently lost.

The application of heuristic 4 enables to link the events representing the trigger and postconditions of `Calculate` by the → operator in order to build the operation specification. In accordance with heuristic 5 the associate process has *Calculate* as name. From there a first draft of `PRECON` specification can be constructed:

component PRECON

> **port** AcquisitionInput = receive?si → AcquisitionInput
>
> **port** ALARMOutput = ...
>
> **computation** = Calculate ... **where**
> Calculate = AcquisitionInput.receive?si → calculationDone

Consider now the second operation `DetecFault` whose KAOS specification is:

Operation `DetectFault`

> **Input** f: Fault, l: Location
>
> **Output** /
>
> **DomPre** ¬ 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

Comparing the trigger condition of `Calculate` and the postcondition of `Calculate` one can note that heuristic 7 applies in this case since

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

`DetectFault` should therefore be applied directly after `Calculate`. A part of the trigger condition `DetectFault` matches the postcondition of `Calculate` and will therefore be expressed by the same event. Heuristic 2 applies to $Occurs(f,l)$ since it is clearly a property of the environment. Occurs(f,l) will thus be expressed by a condition using the *when* operator. Heuristic 1 enables to express the postcondition *Detected*(f,l) as an event. Note that since the pull dataflow connector has been chosen, the event of detect a fault corresponds to the *DataReady* event of the connector. In fact, the interaction between `PRECON` and `ALARM` is structured as follows: when `PRECON` detects a fault, it notifies `ALARM` that will subsequently ask for the corresponding fault information. The event *Detected*(f,l) has therefore to be sent on the port *AlarmOutput*. Heuristics 4 and 5 allow to complete the specification. The specification of `PRECON` can be rewritten as follows with the added operation:

component PRECON

port AcquisitionInput = receive?si → AcquisitionInput

port ALARMOutput = $\overline{ALARMOutput.Detected(f,l)}$

computation = Calculate **where**

$$\text{Calculate} = \begin{cases} AcquisitionInput.receive?si \rightarrow calculationDone \\ \rightarrow DetectFault \end{cases}$$

$$\text{DetecFault} = \begin{cases} \overline{ALARMOutput.Detected(f,l)} \rightarrow \ldots & \textbf{when} Occurs(f,l) \\ \textbf{computation} & \textbf{when} \neg Occurs(f,l) \end{cases}$$

Since the two following operations – `SwitchFaultStatusOn` and `Switch-FaultStatusOff` are very similar they will be handled simultaneously. Their specicification is given by:

Operation `SwitchFaultStatusOn`
Input f: Fault, l: Location, PowerPlant
Output PowerPlant/FaultStatus
DomPre PowerPlant.FaultStatus = off
DomPost PowerPlant.FaultStatus =on
ReqTrig For FaultStatusUpdated
Detected(f,l)

Operation `SwitchFaultStatusOff`
Input f: Fault, l: Location, PowerPlant
Output PowerPlant/FaultStatus

DomPre PowerPlant.FaultStatus = on
DomPost PowerPlant.FaultStatus = off
ReqPre For FaultStatusUpdated
¬ Detected(f,l)

Heuristic 7 applies to `DetectFault` and `SwitchFaultStatusOn` since the postcondition and the trigger conditions are equivalent. These two operations will therefore be composed using the → operator in Wright. Heuristic 6 applies to `SwitchFaultStatusOn` and `SwitchFaultStatusOn`, so stating that there is a sequential order between them. However those two operations are not successive since heuristic 7 does not apply. In Fact to switch the fault status off, it must have been turned on before and vice versa. Another heuristic must be applied to compose this operations with the three others that are successive. Heuristic 9 applies here since the precondition of `Calculate` refers to the environment and the operations will thus used [] to be composed in Wright. The preconditions of these two operations refer to the environment and will thus be expressed as conditions as stated in heuristic 2. Proceeding in a similar way than for the other operations, the complete specification of PRECON becomes:

component PRECON

port AcquisitionInput = receive?si → AcquisitionInput

port ALARMOutput = $\overline{ALARMOutput.Detected(f,l)}$

computation = Calculate [] SwitchFaultStatusOff **where**

$$\text{Calculate} = \left\{ \begin{array}{l} AcquisitionInput.receive?si \rightarrow calculationDone \\ \rightarrow DetectFault \end{array} \right.$$

$$\text{DetecFault} = \left\{ \begin{array}{ll} \overline{ALARMOutput.Detected(f,l)} & \\ \rightarrow SwitchFaultStatusOn & \textbf{when } Occurs(f,l) \\ \textbf{computation} & \textbf{when} \neg Occurs(f,l) \end{array} \right.$$

$$\text{SwitchFaultStatusOn} = \left\{ \begin{array}{ll} FaultStatusOn & \\ \rightarrow \textbf{computation} & \textbf{when } FaultStatusOff \\ \textbf{computation} & \textbf{when } FaultStatusOn \end{array} \right.$$

$$\text{SwitchFaultStatusOff} = \left\{ \begin{array}{ll} FaultStatusOff & \\ \rightarrow \textbf{computation} & \textbf{when } \quad FaultStatusOn \\ & \quad \wedge \neg Occurs(f) \end{array} \right.$$

Now that `PRECON` has been specified, the same has to be done with `ALARM`. ALARM has to perform three operations:

1. RaiseAlarm
2. SwitchAlarmStatusOn

3. SwitchAlarmStatusOff

The first operation specification is given by:

Operation `RaiseAlarm`

> **Input** fi: FaultInformation
>
> **Output** a: Alarm
>
> **DomPre** ¬ Raise(fi,a)
>
> **DomPost** Raise(fi,a)
>
> **ReqTrig For** AlarmRaisedWhenFaultInformationTransmitted
> ¬ Raise(fi,a) $\mathbf{S}_{=1s}$ Transmitted(fi,PRECON, ALARM) ∧ ¬ Raise(fi,a)
>
> **PerformedBy** ALARM

This operation is very similar to the the `Calculate` operation of PRECON and all what has been said for `Calucalte` also applies to `RaiseAlarm`. Figure 3.5 shows the dataflow connector between `PRECON` and `ALARM`. So heuristic 3 states that there will be an event e of the form $e?fi$ in `ALARM` specification. This events corresponds to $Transmitted(fi, PRECON, ALARM)$ in `RaiseAlarm` specification. An alarm can only be raised once as it is clearly stated that the raising of an alarm occurs always after the reception of a fault information and that a fault information is never received twice. A first draft of `ALARM` specification can be derived:

component ALARM

> **port** PRECONInput = Detected(f,l) → $\overline{request!f}$ → receive?fi → PRECONInput
>
> **computation** = RaiseAlarm . . . **where**
> RaiseAlarm = PRECONInput.receive?fi → Raise(fi,a)

`SwitchAlarmStatusOn` and `SwitchAlarmStatusOff` are the dual of `SwitchFaultStatusOn` and `SwitchFaultStatusOff` for `ALARM`. Here are the specifications:

Operation `SwitchAlarmStatusOn`

Input a: Alarm, fi: FaultInformation, PowerPlant

Output PowerPlant/AlarmStatus

DomPre PowerPlant.AlarmStatus = off

DomPost PowerPlant.AlarmStatus = on

ReqTrig For AlarmStatusUpdated
Raise(fi,a)

PerformedBy ALARM

Operation `SwitchAlarmStatusOff`
Input a: Alarm, fi: FaultInformation, PowerPlant
Output PowerPlant/AlarmStatus
DomPre PowerPlant.AlarmStatus = on
DomPost PowerPlant.AlarmStatus = off
ReqPre For AlarmStatusUpdated
¬ Raise(fi,a)
PerformedBy ALARM

Because of the similarity with `SwitchFaultStatusOn` and `SwitchFaultStatusOff` the derivation process is essentially the same. Important points are the sequentiality of `SwitchAlarmStatusOn` and `SwitchAlarmStatusOn` confirmed by heuristic 6 and the direct succession of `RaiseAlarm` and `SwitchAlarmvStatusOn` confirmed by heuristic 7. The final Wright specification of `ALARM` is:

component ALARM

port PRECONInput = Detected(f,l) → $\overline{request!f}$ → receive?fi → PRECONInput

computation = RaiseAlarm [] SwitchAlarmStatusOff **where**
RaiseAlarm = PRECONInput.receive?fi → Raise(fi,a) → SwitchAlarmStatusOff

$$\text{SwitchAlarmStatusOn} = \begin{cases} AlarmStatusOn & \\ \rightarrow \textbf{computation} & \textbf{when } AlarmStatusOff \\ \textbf{computation} & \textbf{when } AlarmStatusOn \end{cases}$$

$$\text{SwitchAlarmStatusOff} = \begin{cases} AlarmStatusOff & \\ \rightarrow \textbf{computation} & \textbf{when } AlarmStatusOn \\ & \wedge \neg Raise(fi,a) \end{cases}$$

The architecture fragment demonstrates the use of the different derivation heuristics presented in Section 3.2. They are all but perfect but nonetheless provide a useful help. The heuristics dealing with order of application are particularly important, especially heuristic 7 that defines a criterion of direct succession between two operations. Since this heuristic is based on the milestone refinement pattern and on the bounded achieve operationalization pattern it has been extensively used for this system. This example also enables to pinpoint the weaknesses of Wright. No account is taken of time and since all the descriptions are formalized in linear temporal logic there will inevitably be a loss of semantic. A solution to that problem could be to define an additional *constraints* field to the component specification in order to specify formally the time constraints for example. However it will not be used for formal reasoning. For example the following constraint

can be added to component `ALARM` to state that an alarm has to be raised within one second after the reception of the fault information.

constraints $\forall$ fi:FaultInformation, $\exists$!a:Alarm
Transmitted(fi,PRECON,ALARM) $\Rightarrow \Diamond_{\leq 1s}$ Raise(fi,a)

3.4 Making Architectural Patterns Further Precise

The description of pattern has been identified as a weak point in the KAOS method. This section aims at correcting part of this weakness. Wright will be used to describe precisely the structural and behavioral effects of pattern application. Two patterns are studied: the fault-tolerant communication pattern and the observer pattern, as they are the only applied to the power plant supervisory system. This allows the validation of the pattern description on a real example although the intent of the description is to be general and applicable to any system.

3.4.1 The Fault-Tolerant Communication Pattern

General Considerations on Fault Tolerance

The aim of this work is certainly not to focus on fault tolerance but some notions are nevertheless useful. This subsection will explain the foundations of fault-tolerant distributed computing and set up the underlying model and hypotheses used further. The interested reader can find a summary in [17].

Fault tolerance aims at making distributed systems more reliable by handling faults in complex computing environments. There is now an increasing demand for *dependable* systems, i.e., systems with quantifiable reliability properties. This is particularly true in safety and security-critical applications.

The underlying model used for fault tolerance is the *asynchronous system model*. A distributed system is asynchronous if there is no bound on message delay, clock drift, or the time necessary to execute a step. So no time assumptions will be made whatsoever.

It is also important to clearly state what kind of faults will be tolerated by the pattern. Faults are traditionally grouped in fault classes or fault models. Well-known examples are the *crash failure model* (in which processors simply stop executing at a specific point in time) or *Byzantine* (in which processors

may behave in arbitrary, even malevolent, ways). The faults handled by the pattern belong to the crash failure model.

Systems are characterized by two major properties: safety and liveness. Informally, the *safety* property states that some specific "bad thing" never happens. It describes the set of "legal" system configurations, i.e, the *invariant*. The *liveness* property claims that some "good thing" will eventually happen during system execution. A common example of liveness is *termination*. For formal definitions, please refer to [17].

Different forms of Fault-Tolerance have been defined according to how the safety and liveness properties of the distributed system hold over time in the presence of faults. If a program A still satisfies both its safety and liveness properties in the presence of faults from a specified fault class F, then it is said that A is *masking* fault tolerant for fault class F. This is the strictest, most costly and most desirable form of fault tolerance because the program is able to tolerate the faults transparently. This is the form of fault tolerance this pattern will achieve. The other forms of fault tolerance are summarized in Table 3.2[3].

	live	not live
safe	masking	fail safe
not safe	nonmasking	none

Table 3.2: Four Forms of Fault Tolerance

The notion of redundancy is omnipresent in the field of fault tolerance and it arises for a simple reason: redundancy is a necessary condition to fault tolerance. Two forms of redundancy can be distinguished, namely in *space* and in *time*. Redundancy in space refers to the superfluous part of the state of a system, i.e., states never reached when faults do not occur. It is usually achieved by supplying a component more than once. It will be the case in this pattern. Redundancy in time refers to superfluous state transitions of a system, i.e., the superfluous work it performs. Practical examples include *roll-back recovery* or *reset* procedures. Considering the fault handled by the pattern – process crash – the second one will be used.

Pattern description

Introduction The aim of the pattern is to achieve masking fault-tolerance, i.e., preserve safety and liveness. The pattern will proceed in a stepwise

[3] *live* means satisfying the liveness property and *safe* means satisfying the safety propery

manner, achieving first nonmasking fault tolerance and then masking fault tolerance. This approach is similar to the one described in [3].

The first stage will transform an intolerant component into a nonmasking one. The fault class considered here is the process crash. Ensuring liveness for a certain fault class means that when a fault of that class occurs the component will eventually recover from this fault. So, the pattern has to ensure that in case of a process crash the process will be restarted. This is the *correction* phase.

The second stage will make component masking tolerant. The aim of this transformation is to ensure safety, i.e., preventing the violation of the invariant. This can only be done via the fault detection, i.e., the *detection* phase.

Important remarks have to been made at this point. Due to the underlying model – the asynchronous system model – the problem of the detection of a process crash cannot be solved deterministically. Intuitively, it is because it is impossible to determine whether a process has actually crashed or is only "very slow".

Various approaches have been used to solve this problem. Chandra and Toueg [7] proposed to use what they called *unreliable failure detectors*, a detection mechanism that can make mistakes. An alternative approach, taken by the Isis team [25], is based on the assumption that failure detectors rarely make mistakes. In those cases where a correct process p is falsely suspected by the failure detector, p is effectively forced "to crash". The detector forces the system to conform to its view. From the application's point of view, this failure detector looks perfect. This is the approach followed here.

One will concentrate on abstract concepts rather than giving some specific implementations. The purpose is indeed to help the programmer but not to substitute to him.

Description A first glance of the pattern is presented in Figure 3.6. As previously mentioned, the pattern application encompasses two phases, correction and detection. Each of them will be described further.

Stage 1: Correction phase The aim of this phase is to correct the occurring faults. To do so, a mechanism of recovery should be introduced. As the fault considered in this case are process crashes, the designated mechanism is naturally *reset procedures*. However the problem with reset procedures is the time it takes. Some systems can not afford any delay especially when some time constraints have to be ensured. It is so desirable that the

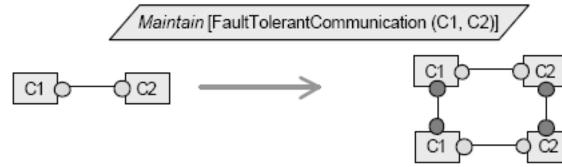

Figure 3.6: Fault-tolerant communication pattern

system goes on with working without any delay. To this end a copy of the component is introduced. Whenever a process crashes, its copy should take on its role. It is important that only one component should be working at a time. After a process has been reset it remains inactive until its copy crashes.

The resulting system has so the following properties.

1. *Introduction of reset procedures*
   A reset mechanism should be introduced to restart any component that may have crashed.

2. *Introduction of copies of components*
   An exact copy of the components between which the communication has to become fault-tolerant is added to the system. The two copies have exactly the same capabilities, i.e, the same specifications in terms of their ADL descriptions.

3. *Switch in case of failure*
   When, for whatever reasons, a component goes down, its copy should take on its role. This change of operating component should not result neither in any error of treatments nor in any loss of information. The crashed component should also be reset.

4. *One at a time*
   In order to avoid duplicate treatments and so on, only one of the two copies should work at a time.

Let's try now to introduce transformations to generics ADL component specifications to ensure those four properties.

First a reset mechanism has to be added to components. This is not as simple as it seems. If a process has crashed, it will not reply to any solicitation from the outside. In fact what should be done is killing the process and

restart it. This leads to some difficulties. The connectors through which the process communicates have to remain active in order not to disturb the surrounding processes. Moreover once the process restarted it should communicate through those connectors. In an aim of simplicity it will be considered that some reset procedure exists and works even if the process has crashed saving to deal with all the implementation aspects. The specifications can be modified as followed.

component C

  port ...
  computation = compute

component newC

  port ...
  computation = compute [] reset → **computation**

The introduction of copies is straightforward. As the specifications of the two copies are strictly equal, an extra instance of the component has simply to be added.

instances x: C1
  y: C2

instances x_1, x_2: C1
  y_1, y_2: C2

The last two properties are tightly linked and so will be the transformations introduced on the system. In order to have two copies of C1 and C2 and if only one copy can operate at a time, the state of the component must be stored in some way in the component specification. A component can be either passive or active. This will be modeled by adding two external events to which the component will react. Those two events are simply *wakeUp* and *sleep*. A component should never be placed in an inactive state if it did not crash before. So this change of state should take place after the reset procedure. The modification can be generalized as follows. The following modification simply states that either the component operates normally or, after crashing, it receives a *sleep* event and will not go on with its duty unless it receives a *wakeUp* event.

component C

port ...

computation = compute [] reset → **computation**

component newC

port ...

computation = compute [] reset → sleep → wakeUp → **computation**

Afterward it should be ensured that the switch in active component only takes place in case of failure, that any crashed component will be reset and that only one copy is working at a time. So to say, the failure should first be corrected and that correction should not be visible to the surrounding components. The inherent problem of restarting a component is that it takes time and that time delay might not be affordable. As soon as the reset procedure has been called on the crashed component, its copy should be made active. So the failure triggers first the reset of the component and then the change of active component. Once the crashed component reset it receives a *sleep* signal signifying that it is now the inactive component. One could suggest to first wake up the safe component before restarting the other one. But resulting from the asynchronous nature of the underlying model, it is impossible to detect whether or not a component has crashed. Doing that it would be possible to have both copies of the components working at the same time. Resulting from those considerations, a preliminary definition of the connector linking the two copies can be introduced.

connector copyConnect

role $\text{Copy}_{1,2}$

glue $= (Copy_1.failure \rightarrow \overline{Copy_1.reset} \rightarrow \overline{Copy_2.wakeUp} \rightarrow \overline{Copy_1.sleep})$
$[](Copy_2.failure \rightarrow \overline{Copy_2.reset} \rightarrow \overline{Copy_1.wakeUp} \rightarrow \overline{Copy_2.sleep})$

With these transformations the four properties stated above hold. This completes the first stage of the method – the corrective one.

Stage 2: Detection phase Now that correction mechanisms have been introduced, mechanisms to detect the occurrence of faults need to be defined. One should remember that the problem of detecting process crashes cannot be solved deterministically. This impossibility results from the inherent difficulty of determining whether a system has actually crashed or is only “very slow”. The chosen approach to solve this problem is to

force any suspected process to reset, based on the assumption that the fault detector rarely makes mistakes. The suspicion of a process will be based on time-outs. The choice of an adequate time-out value is of great importance for the global performance of the system. The value should not be too small to prevent any working process from resetting unnecessarily while it should not be too long in order not to slow down all the system when a process has effectively crashed.

Two main properties must be achieved in this phase. First faults have to be detected and secondly all what has been made so far must be transparent for all the surrounding components. They should not be aware neither of the presence of two copy components nor of any process crash nor of any switch in active components .

A third one states there can be no precedence between the two copies of a component.

Those properties can be summarized as followed.

1. *Introduction of fault detection mechanism*
   A mechanism to detect process crashes should be added. One should note that this detection mechanism could suspect correctly working processes.

2. *No precedence between the two copies*
   There is no master and slave component. The two copies are exactly equal. This equality results in the following policy: the switch of operating component occurs only in case of failure. When a component recovers from a failure it becomes inactive.

3. *Transparency of switching* The switch in the operating component should be done in a completely transparent way to other interacting components. They should not even notice that a switch occurred. Moreover the surrounding components should not be aware of the presence of duplicate copies of the component they interact with.

As in the previous subsection, those properties will be translated in terms of concrete transformations to apply to the ADL description of components.

The introduction of a fault detection mechanism leads to some problems. As previously mentioned the classic solution consists of sending messages to the processes and if they do not answer within a certain time-out value they are suspected of having crashed. The difficulty arises from the fact that Wright does not support such time constraints. However it is possible to specify it using the *when* operator in a very understandable way. First the

connector between the two copies has to send the message to the copies. Second some answering procedure should be added to the components to say “Yes I'm alive”. The modified specifications are given below:

connector copyConnect

role $\text{Copy}_{1,2}$

glue $= \text{Copy}_1$

$$\text{Copy}_1 = \overline{Copy_1.isAlive} \rightarrow \begin{cases} Copy_1 & \textbf{when } Copy_1.ImAlive \text{ within t s} \\ Copy_1Failure & \textbf{when } \neg Copy_1.ImAlive \text{ within t s} \end{cases}$$

$$\text{Copy}_2 = \overline{Copy_2.isAlive} \rightarrow \begin{cases} Copy_2 & \textbf{when } Copy_2.ImAlive \text{ within t s} \\ Copy_2Failure & \textbf{when } \neg Copy_2.ImAlive \text{ within t s} \end{cases}$$

$$\text{Copy}_1\text{Failure} = \begin{cases} \overline{Copy_1.reset} \rightarrow \overline{Copy_2.wakeUp} \\ \rightarrow \overline{Copy_1.sleep} \rightarrow Copy_2 \end{cases}$$

$$\text{Copy}_2\text{Failure} = \begin{cases} \overline{Copy_2.reset} \rightarrow \overline{Copy_1.wakeUp} \\ \rightarrow \overline{Copy_2.sleep} \rightarrow Copy_1 \end{cases}$$

component newC

$$\textbf{port} \quad \text{Copylink} = \begin{cases} isAlive \rightarrow \overline{ImAlive} \rightarrow \textbf{Copylink} \\ [] \; reset \rightarrow sleep \rightarrow wakeUp \rightarrow \textbf{Copylink} \end{cases}$$

$$\textbf{computation} = \begin{cases} compute \\ [] \; reset \rightarrow sleep \rightarrow wakeUp \rightarrow \textbf{computation} \\ [] \; Copylink.isAlive \rightarrow \overline{Copylink.ImAlive} \rightarrow \textbf{computation} \end{cases}$$

One can note in the previous specification it has been arbitrarily specified that Copy_1 is the first to be active. As both copies are strictly equivalent this does not invalidate the no precedence policy between the copies.

When considering transparency, things are getting far more complicated. Transparency means that first every component interacting with a “double” component should not even notice this fact and secondly the switch in operating component should also not be noticed. The solution adopted is a “forked” connector. In fact all the interaction with a “double” component will be made with the two at the same time, as it was previously ensured only one will respond as the other will be asleep. It could appear a bit strange but it is in fact a simple and practical solution. For the components it is exactly the same situation as before. They just do not know that they use now a “forked” connector. Of course the specification of the role of the connectors can not change otherwise it will be visible to the component because the role has to be compatible with the port to have a working specification.

So only the glue will be modified. So two new kinds of connectors will be introduced. A *right-forked* connector for the components interacting with a "double" component that are not "doubled" themselves, a *left-forked* for the double components interacting with a single component and a *double-forked* connector to link two "double" components. The modified connectors are defined as follows:

connector oldConnect

role r1

role r2

glue $= \text{r1.eventA} \rightarrow \overline{r2.eventB} \rightarrow \ldots \rightarrow$ **glue**

connector right-forked

role r1

role Copy_1

role Copy_2

glue $= \text{r1.eventA} \rightarrow (\overline{Copy_1.eventB} \parallel \overline{Copy_2.eventB}) \rightarrow \ldots \rightarrow$ **glue**

connector left-forked

role r1

role Copy_1

role Copy_2

glue $= (\text{r1Copy}_1\text{.eventA} \; [] \; \text{r1Copy}_2\text{.eventA}) \rightarrow \overline{r2.eventB} \rightarrow \ldots \rightarrow$ **glue**

connector double-forked

role r1Copy_1

role r1Copy_2

role r2Copy_1

role r2Copy_2

glue $= (\text{r1Copy}_1\text{.eventA} \; [] \; \text{r1Copy}_2\text{.eventA}) \rightarrow (\overline{r2Copy_1.eventB} \parallel \overline{r2Copy_2.eventB}) \rightarrow \ldots \rightarrow$ **glue**

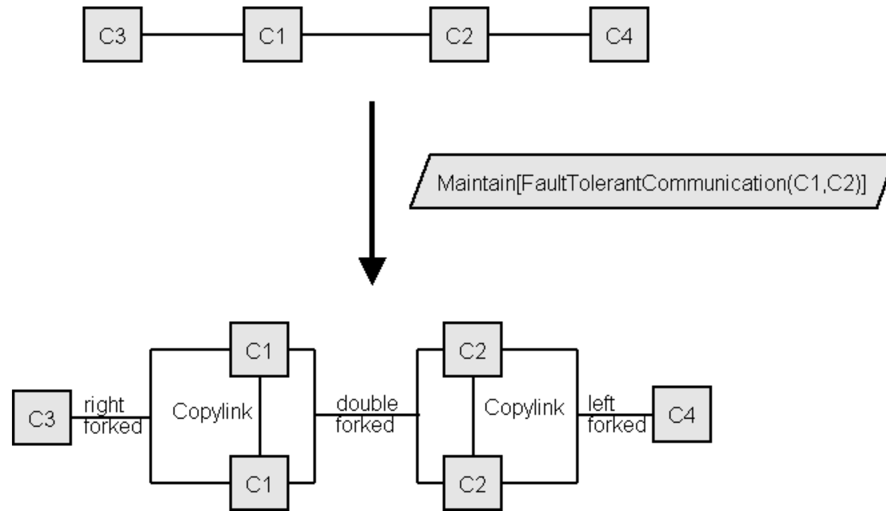

Figure 3.7: Fault-tolerant communication pattern Transformation rule

The introduction of forked connectors changes the specifications as follows. If it is the source of an event which is double, the event could be notified by either one of those two copies depending on the active one. The event is so marked as a source of each copy and a *deterministic choice* operator is added between those two events. If the destination of an event is double, the event is notified to the two copies in a parallel way. The *parallel composition* operator is so used. For double-forked connectors a similar procedure is used for both sides of the connectors.

A general graphical view of the transformation rule, including the forked connectors, is shown in Figure 3.7.

Application of the pattern on `PRECON` and `ALARM`

The pattern was applied on `PRECON` and `ALARM` as prescribed by the KAOS method. Starting from the Wright specifications built in Section 3.3, all the modifications and additions described in this section were made. The resulting fragment of architecture was obtained without any difficulty. The Wright specifications of the initial and of the resulting pattern-refined architecture fragment can be found in Appendix C.1.

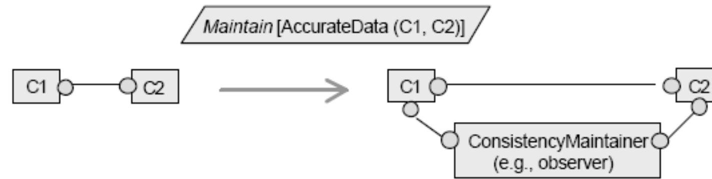

Figure 3.8: Observer pattern

3.4.2 The Observer Pattern

The aim of the Observer pattern is to maintain consistency between related objects. When one of these objects undergoes some change, all its dependents are notified and updated automatically. This should be done by minimizing interdependencies between objects, so achieving *Low Coupling*.

One classical use of this pattern is the decoupling of the presentational aspects from the underlying application data. Consider for example a set of numerical data. This information can be viewed from different ways, e.g., a spreadsheet, a bar chart or a pie chart. All these views are related to the same data and should thereby be coherent and consistent all together. If any modification is introduced via the spreadsheet for example, it should be reflected on the related views.

The Observer pattern distinguishes two main concepts : the *subject* and the *observers*. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response each observer will query the subject to synchronize its state with the subject's state.

Description

A graphic overview of the pattern is presented in Figure 3.8 such as suggested in [27].

Before explaining how the pattern is applied it will first be focused on the concepts and mechanisms involved in order to have a general idea of what properties should have the resulting architecture. This description will be based on the Observer pattern description made by Gamma *et* al in [15]. There are nonetheless adaptations so as to make it applicable to Wright.

The pattern is composed of two different components types: the subject and the observers. The subject is what makes the observers dependent from each other. It is the common data that are either accessed or updated by

the observers. The subject knows its different observers while the observers do not know each other. This enables to minimize the interdependencies between observers so achieving low coupling. Figure 3.9 shows the structure of the observer pattern, that is, how the subject and the observers are interconnected together. Note that the observers are not linked together.

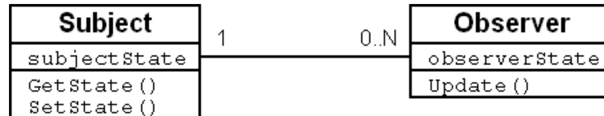

Figure 3.9: Observer structure

The collaboration between the different participants is organized as follows. Whenever the subject undergoes some change, it notifies its dependent observers since there state could be inconsistent. It is thereafter their duty to query the subject for information. The observers use this information to synchronize their state with the subject's state. Figure 3.10 shows an example of interaction scenario between a subject and its two observers.

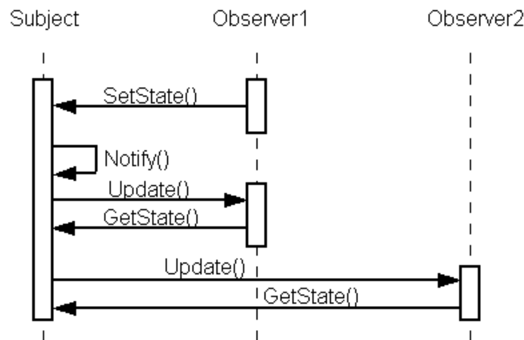

Figure 3.10: Observer behavior

The idea of the pattern is to prevent from having inconsistent data due to scattered modification among the different components by centralizing the data itself, the update and the access procedure. Various steps are required to achieve it.

1. *Identification of the subject and of the observers*
   Both the data updated by the different components and the components themselves have to be identified.

2. *Creation of the subject component*

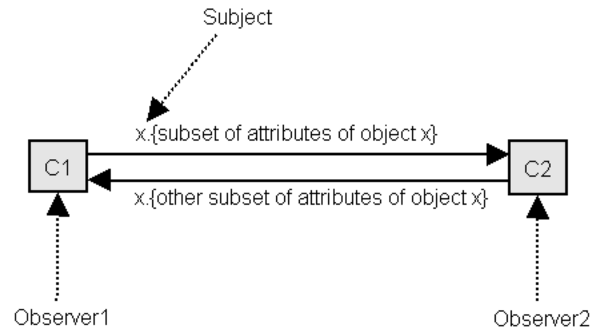

Figure 3.11: Identification of the subject and the observers

Once the subject has been identified, the subject component can be created. More accurately, a generic subject component is instantiated to suit the particular needs of the data. The subject component should define in its interface a procedure to enable observers to update and to obtain its state.

3. *Adaptation of the observers behavior*
   Each time an observer modifies the data, it has to update the subject state and each time it receives a notification from the subject it has to get the new state from the object

4. *Suppression of the old synchronization mechanisms*
   The old interactions between the different observers in order to synchronize their data have to be deleted since this is now done through the subject.

5. *Addition of new connectors between the subject and the observers*
   In order to enable the subject and the observers to communicate, new connectors have to be added.

The identification of the subject and observers is done without any real difficulty from the initial fragment. As shown in Figure 3.8 the two observers are C1 and C2. The subject is the common data updated by both C1 and C2. It is indicated on the graphical representation by two dataflow connectors, each one carrying different attributes of the same object. This is illustrated in Figure 3.11.

The subject component should have three operations: one to allow observer to update its state, one to allow observer to get its state and one to notify all its observers when some modifications has occured. The Update

operations should therefore be directly followed by the notify operation. The component subject can be defined in Wright as follows:

component Subject

$$\textbf{port}\ \text{Observer}_{1,2}\text{Link} = \begin{cases} SetState?x \rightarrow \overline{Notify} \rightarrow \textbf{Observer}_{\mathbf{1,2}}\textbf{Link} \\ [\,]\ GetState?x \rightarrow \overline{SendState!y} \rightarrow \textbf{Observer}_{\mathbf{1,2}}\textbf{Link} \end{cases}$$

computation = SetState → Notify [] GetState **where**

SetState=(Observer$_1$.SetState?x [] Observer$_2$.SetState?x)

$$\text{Notify} = (\overline{Observer_1.Notify} \parallel \overline{Observer_2.Notify}) \rightarrow \textbf{computation}$$

$$\text{GetState} = \begin{cases} (Observer_1.GetState?x \rightarrow \overline{Observer_1.SendState!y} \\ \ \rightarrow \textbf{computation}) \\ [\,](Observer_2.GetState?x \rightarrow \overline{Observer_2.SendState!y} \\ \ \rightarrow \textbf{computation} \end{cases}$$

Thereafter the observers behavior has to be modified so as to ensure two things. First every update on the observer state is effectively reflected on subject state. Secondly every notification of a change received from the subject leads to a query to the subject in order to synchronize observer state and subject state. In order to support these new characteristics, the Wright component of the observer can be modified as follows:

component Observer

port ToOtherObserver ...

computation = compute [] §**where**

compute = ... → Modify(x) → ...

component NewObserver

port ToOtherObserver ...

$$\textbf{port}\ \text{SubjectLink} = \begin{cases} \overline{SetState!x} \rightarrow \textbf{SubjectLink} \\ [\,]\ Notify \rightarrow \overline{GetState!x} \rightarrow SendState?y \rightarrow \textbf{SubjectLink} \end{cases}$$

computation = compute [] Update **where**

$$\text{compute} = \ldots \rightarrow \text{Modify(x)} \rightarrow \overline{SubjectLink.SetState!x} \ldots$$

$$\text{Update} = \begin{cases} Subject.Notify \rightarrow \overline{SubjectLink.GetState!x} \\ \rightarrow SubjectLink.SendState?y \rightarrow SynchronizeValue \\ \rightarrow \textbf{computation} \end{cases}$$

All the communications between the observers aiming at keeping the state of the subject consistent and coherent must be deleted. This may result in the suppression of a connector and therefore of a port in the Wright

specification. The two components are so made more independent from each other. The observer pattern achieves an other non-functional requirement than `Maintain[AccurateData(C1,C2)]`, namely `Maintain[LowCoupling(C1,C2)]`. Assuming that the only communication present between the two observers is for synchronization purpose, the Wright specification can be modified as follows:

component NewObserver

port ToOtherObserver ...

$$\textbf{port}\ \text{SubjectLink} = \begin{cases} \overline{SetState!x} \rightarrow \textbf{SubjectLink} \\ [\,]\ Notify \rightarrow \overline{GetState!x} \rightarrow SendState?y \rightarrow \textbf{SubjectLink} \end{cases}$$

computation = compute [] Update **where**

$$\text{compute} = \begin{cases} \ldots ToOtherObserver.receive?x \ldots \rightarrow Modify(x) \\ \rightarrow \overline{SubjectLink.SetState!x} \rightarrow \overline{ToOtherObserver.Send!x} \ldots \end{cases}$$

$$\text{Update} = \begin{cases} Subject.Notify \rightarrow \overline{SubjectLink.GetState!x} \\ \rightarrow SubjectLink.SendState?y \rightarrow SynchronizeValue \\ \rightarrow \textbf{computation} \end{cases}$$

component NewObserver

$$\textbf{port}\ \text{SubjectLink} = \begin{cases} \overline{SetState!x} \rightarrow \textbf{SubjectLink} \\ [\,]\ Notify \rightarrow \overline{GetState!x} \rightarrow SendState?y \rightarrow \textbf{SubjectLink} \end{cases}$$

computation = compute [] Update **where**

$$\text{compute} = \ldots \rightarrow \text{Modify(x)} \rightarrow \overline{SubjectLink.SetState!x} \ldots$$

$$\text{Update} = \begin{cases} Subject.Notify \rightarrow \overline{Subject.GetState!x} \\ \rightarrow Subject.SendState?y \rightarrow SynchronizeValue \\ \rightarrow \textbf{computation} \end{cases}$$

Finally, new connectors have to be introduced in order to link the subject and the observers. Their Wright description is given by:

connector Observer-Subject Link

$$\textbf{role}\ \text{Observer} = \begin{cases} \overline{SetState!x} \rightarrow \textbf{Observer} \\ [\,]\ Notify \rightarrow \overline{GetState!x} \rightarrow SendState?y \rightarrow \textbf{Observer} \end{cases}$$

$$\textbf{role}\ \text{Subject} = \begin{cases} SetState?x \rightarrow \overline{Notify} \rightarrow \textbf{Subject} \\ [\,]\ GetState?x \rightarrow \overline{SendState!y} \rightarrow \textbf{Subject} \end{cases}$$

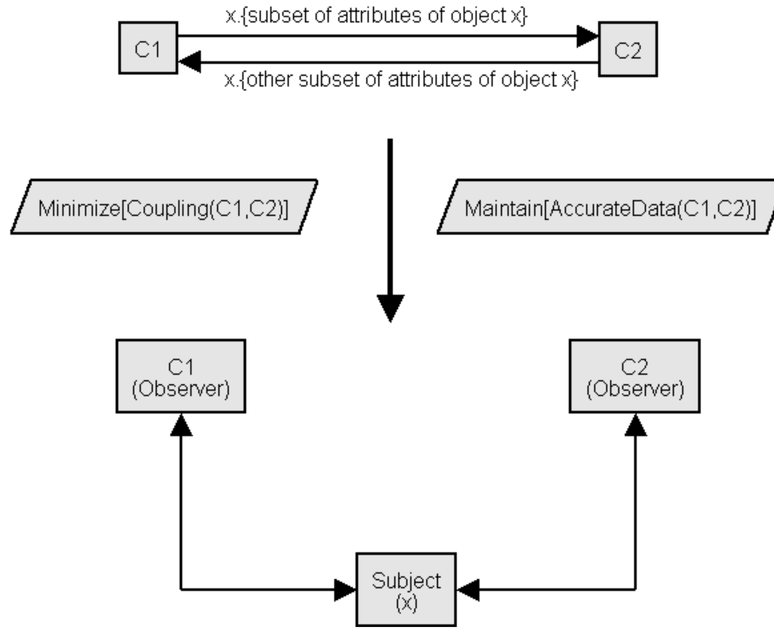

Figure 3.12: Observer pattern transformation rule

$$\textbf{glue} = \begin{cases} Observer.SetState!x \\ \rightarrow \overline{Subject.SetState?x} \rightarrow Notify \quad \textbf{where} \\ []Notify \end{cases}$$

$$\text{Notify} = \begin{cases} Subject.Notify \rightarrow \overline{Observer.Notify} \\ \rightarrow Observer.GetState!x \rightarrow \overline{Subject.GetState?x} \\ \rightarrow Subject.SendState!y \rightarrow \overline{Observer.SendState?y} \rightarrow \textbf{glue} \end{cases}$$

The observer pattern as described here is a bit different from what was suggested in Figure 3.8 since the connector between C1 and C2 is deleted (only if its only purpose is to transmit the object x). This enables to achieve low coupling between C1 and C2 in addition to the initial non-functional requirement (`Maintain[AccurateData(C1,C2)]`). The graphical representation of the transformation rule is therefore slightly different and is presented in Figure 3.12.

Application of the pattern on `Acquisition Unit` and `IMS`

The pattern was applied on `Acquisition Unit` and `IMS` without any particular difficulty. The specification of the initial architecture fragment together

with the architecture specification resulting from the application of the observer pattern can be found in Appendix C.2.

3.5 Discussion

In this chapter, the use of Wright has been examined in order to derive a more precise architecture. The two goals pursued were the addition of a behavioral view to the architecture resulting from the KAOS method and to describe the architecture parts resulting from the application of patterns.

The first objective was reached through the introduction of translation mechanisms from KAOS architecture descriptions to Wright specifications. Those mechanisms deal with both the structural and the behavioral aspects. Rules expressed the former while heuristics the latter. This difference of power is caused by two facts:

1. The constructs used by KAOS and Wright dealing with the structure are essentially the same.
2. The behavioral view is only partially present in the KAOS architecture description and the global control flow has to be inferred from operations specification.

Moreover Wright does not support temporal logic and there is inevitably a loss of semantic during the translation. Nevertheless Wright specifications of architecture provide a significantly more precise architecture description. From the addition of behavior specifications, might follow the possibility to validate the architecture with respect to the functional requirements. The animator ProBE[14] available for CSP could be used to visualize the possible execution traces so enabling to check whether the architecture meets the functional requirements.

Two patterns have been examined using Wright: the fault-tolerant communication pattern and the observer pattern. Both the resulting architecture fragment and the transformation process have been described. It is now clear how the application of these patterns modifies the involved components and connectors specification as well as the exact functionalities performed by the added ones.

These two areas of improvement have been validated on the power plant supervisory system. In both cases the results are very encouraging. The defined rules and heuristics provide a significant help to derive the Wright specifications from the KAOS architecture description and the application

of the two patterns does not present any difficulty. The introduced approach seems very productive.

Conclusion

This thesis evaluated and compared two methods for deriving architecture from goal-oriented requirements. The KAOS method and the Preskriptor process were validated via their application on a power plant supervisory system. The system was developed by ENEL, this Italian electricity company and its description was extracted from papers reporting the project.

The conducted analysis shows as main result that both approaches seem effective. The two resulting architectures satisfy all the functional and most of the non-functional requirements. Performance was for example left unhandled in both cases. Some weaknesses were nonetheless present. Neither method produced an architecture description dealing with behavioral aspects. The descriptions were essentially structural while the way the different components interact with each other remained uncovered. Moreover the derivation processes in themselves appear to be a bit hazy in certain aspects. Pattern and styles description with respect to their choice, their applicability and the architecture resulting from their application is up to now only suggested by the KAOS method. The treatment of non-functional requirements in the Preskriptor process is so far only done for particular examples and does not provide sufficient guidance to be really useful.

The two methods differ greatly by their intent and this is reflected by the resulting architectures. On the one hand the KAOS method aims at making the derivation process as automated and clear as possible so as to provide the maximum level of guidance to the architect. It also targets an architectural description as precise and complete as possible in order to bring the maximum support to designers. On the other hand the Preskriptor process lets more freedom to the architect and uses the concept of architectural prescriptions – expressed in the domain language by opposition to classical descriptions expressed in solution language – so as to make the transition from requirements as smooth as possible. The resulting architecture is therefore less constraining and complete, so letting free the designer to choose the low level design.

In order to improve the KAOS method the use of an architecture description language was explored. Wright was chosen for its ability to describe the multiple facets of an architecture and more particularly the behavioral aspect. Moreover it provides formal reasoning capabilities enabling for example to check for absence of deadlock. It has been argued for an integrate rather than a sequential approach where the Wright specification is derived incrementally from the intermediate results produced by the KAOS method. Various mechanisms have therefore been introduced to derive a Wright specification from the abstract data flow architecture. These include a set of rules and heuristics enabling to derive the component behavior from KAOS operations specifications. The one ruling the application order of operations seems particularly useful. The ability of Wright to describe patterns thereafter has been examined. Two pattern examples were inspected in details: the fault-tolerant communication pattern and the observer pattern. Each of them has been described extensively in terms of the architectural transformation involved as well as regarding the resulting architecture. It emerges from their application to the power plant supervisory system that the introduced approach is highly productive.

The work presented in this thesis contributes to both methods studied by confronting them to a real safety-critical system whose size is reasonable. Moreover significant improvements have been made to the KAOS method by integrating the benefits of architectural descriptions languages.

Nonetheless a lot of questions remain open, leaving space for further work.

Although styles description was identified as a weakness of the KAOS method, no improvement has been proposed in this direction. Few styles are available and both their choice and application are so far very qualitative. They should therefore be investigated further.

Proposed improvements on patterns do not address all the problems. When two patterns are applicable, the issue of combining them, provided they can be combined, is still unclear. Do they keep their efficiency toward non-functional requirements satisfaction while they are combined? If not, how to choose the one to apply? Does the order of application matter? Once one pattern has been applied, will the resulting piece of architecture still match the applicability condition of the other? In addition, the description of the applicability condition is up to now very informal. All those concerns need to be studied.

Wright is currently underused since none of its formal reasoning capabilities have been explored so far. Patterns are not supposed to prevent satisfaction of any functional goal they should therefore not modify the

global behavior of the fragment affected by their application. Some further work should include a validation of the introduced patterns with respect to that condition.

More globally, the validation of the architecture with respect to functional requirements should be examined. To this end, the use of the existing animator for CSP (ProBE) should be inspected. The visualization of the possible execution traces could be a mean to check whether the architecture meets the functional requirements.

Finally, the architecture derivation process is so far not supported by any tool. A potential tool should integrate closely with the derivation process, that is, each step should be supported separately. The support provided at each step should include a derivation part as well as a validation part. The derivation part should help the architect both in his choices (e.g., choice of the suitable style/pattern, application of the correct heuristic) and in their application. The validation part should include an animator so as to provide the architect with a graphical mean to explore the system behavior and a model checker to verify the satisfaction of system properties such as deadlock freedom.

Bibliography

[1] Allen, R. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, May 1997.

[2] Allen, R., and Garlan, D. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology 6*, 3 (July 1997), 213–249.

[3] Arora, A., and Kulkarni, S. S. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions On Sofware Engineering 24*, 6 (June 1998), 435–450.

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

[5] Brandozzi, M., and Perry, D. E. Transforming goal oriented requirement specifications into architectural prescriptions. In *Proceedings of STRAW 2001 - From Software Requirements to Architectures* (2001), Castro and Kramer, Eds., pp. 54–60.

[6] Brandozzi, M., and Perry, D. E. Architectural prescriptions for dependable systems. In *Proceedings of ICSE 2002 - International Workshop on Architecting Dependable Systems* (Orlando, May 2002).

[7] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43*, 2 (March 1996), 225–267.

[8] Ciapessoni, E., Mirandola, P., Coen-Porisini, A., Mandrioli, D., and Morzenti, A. From formal models to formally based methods: an industrial experience. *ACM Transactions on Software Engineering and Methodology 8*, 1 (January 1999), 79–113.

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

[10] Coen-Porisini, A., Pradella, M., Rossi, M., and Mandrioli, D. A formal approach for designing corba based applications. In *Proceedings of the 22nd International Conference on on Software Engineering - ICSE'2000* (Limerick, June 2000), ACM Press, pp. 188–197.

[11] Darimont, R. *Process Support for Requirement Elaboration*. PhD thesis, Université catholique de Louvain, 1995.

[12] Darimont, R., and van Lamsweerde, A. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering - FSE'96* (San Fransisco, October 1996), ACM Press, pp. 179–190.

[13] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*. Oxford, England, May 2003. `http://www.fsel.com`.

[14] Formal Systems (Europe) Ltd. *Process Behavior Explorer: ProBE User Manual*. Oxford, England, January 2003. `http://www.fsel.com`.

[15] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns - Elements of reusable object-oriented software*. Addison-Wesley, 1995, pp. 293–299.

[16] Garlan, D. Formal modeling and analysis of software architecture: Components, connectors, and events. In *Formal Methods for Software Architectures*, M. Bernardo and P. Inverardi, Eds., vol. 2804 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003, pp. 1–24.

[17] Gärtner, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys 31*, 1 (March 1999), 1–26.

[18] Kramer, J., Magee, J., and Uchitel, S. Software architecture modeling & analysis: A rigorous approach. In *Formal Methods for Software Architectures*, M. Bernardo and P. Inverardi, Eds., vol. 2804 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003, pp. 44–51.

[19] Letier, E., and van Lamsweerde, A. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference of Software Engineering - ICSE'2002* (Orlando, May 2002), ACM Press, pp. 83–93.

[20] Letier, E., and van Lamsweerde, A. Deriving operational software specifications from system goals. In *Proceedings of the 10th ACM Symposium on the Foundations of Software Engineering - FSE'2002* (Charleston, November 2002), ACM Press, pp. 119–128.

[21] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference - ESEC'95* (September 1995), Springer-Verlag, pp. 137–153.

[22] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992, ch. 3.

[23] Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the c2 style. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering - FSE'96* (1996), ACM Press, pp. 24–32.

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

[25] Ricciardi, A. M., and Birman, K. P. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing* (1991), ACM Press, pp. 341–353.

[26] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[27] van Lamsweerde, A. From system goals to software architecture. In *Formal Methods for Software Architectures*, M. Bernardo and P. Inverardi, Eds., vol. 2804 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003, pp. 25–43.

[28] van Lamsweerde, A., Darimont, R., and Letier, E. Managing conflicts in goal-driven requirements engineering. *IEEE Transaction on Software Engineering 24*, 11 (November 1998), 908–926.

[29] van Lamsweerde, A., and Letier, E. Handling obstacles in goal-oriented requirements engineering. *IEEE Transaction on Software Engineering 26*, 10 (October 2000), 978–1005.

List of Figures

1.1 Preliminary goal graph for the power plant supervisory system 12
1.2 First draft of the object model 13
1.3 Second draft of the object model 13
1.4 Addition of missing goals via WHY elicitation 14
1.5 Addition of missing subgoals via HOW elicitation 15
1.6 Potential agents 15
1.7 Assigned agents, their interfaces and data dependencies . . . 25
1.8 Derived dataflow architecture 26
1.9 Event-based style transformation rule 27
1.10 Style-based architecture 27
1.11 The `NoReadUpNoWriteDown` pattern for condidentiality goals . 29
1.12 The 3 required steps of the Preskriptor process 31
1.13 Step 4 of the Preskriptor process 33

2.1 The ENEL's supervision and control system[8] 36
2.2 Milestone refinement pattern 39
2.3 General Structure of the goal diagram 40
2.4 Refinement of the goal `FaultsDetected` 40
2.5 Refinement of the goal `AlarmCorrectlyManaged` 41
2.6 Communication reliability refinement subtree 42
2.7 Transmission meta-relationship 43
2.8 Bounded achieve opeartionalization pattern 47
2.9 Immediate achieve operationalization pattern 48
2.10 Refinement and obstacles for the goal `FaultDetectedWhen-CalculationDone` 52
2.11 Abstract dataflow architecture 57
2.12 Centralized communication architectural style 58
2.13 Style-based refined architecture 59
2.14 Fault-tolerant refinement pattern 60

2.15 Consistency maintainer refinement pattern 61
2.16 Pattern-based refined architecture 62
2.17 component refinement tree resulting from step 1 63
2.18 Component refinement tree resulting from step 3 67
2.19 Box-and-line diagram 68

3.1 Fragment of an abstract dataflow architecture 84
3.2 Example of interaction scenario for the "push" behavior . . . 85
3.3 Example of interaction scenario for the "pull" behavior 85
3.4 Milestone refinement pattern with corresponding operationalizations 93
3.5 Fragment of the dataflow architecture containing `PRECON` and `ALARM` 96
3.6 Fault-tolerant communication pattern 106
3.7 Fault-tolerant communication pattern Transformation rule . . 112
3.8 Observer pattern 113
3.9 Observer structure 114
3.10 Observer behavior 114
3.11 Identification of the subject and the observers 115
3.12 Observer pattern transformation rule 118

A.1 Functional goal diagram 132
A.2 Non-functional goal diagram 143
A.3 ObjectDiagram 144
A.4 Responsibility assignment of the different agents 150
A.5 Agent context diagram 151
A.6 Modifications brought to the NFG diagram after the obstacle analysis 163

List of Tables

1.1 Comparison between C2, Darwin and Wright 23
1.2 Mapping KAOS entities to APL entities[4] 31

2.1 Comparison between the resulting architectures 70

3.1 Cases where Op_1 and Op_2 must be applied sequentially . . . 91
3.2 Four Forms of Fault Tolerance 104

Appendix A

KAOS Specifications

A.1 Goal specifications

A.1.1 Functional goals

The functional goals are listed following an alphabetical order. Figure A.1 shows the functional goal diagram.

1. **Goal** `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

2. **Goal** `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)$\wedge$ Representation(ai,a)
    $\wedge$ Concerns(ad,fi,ai) $\wedge$ Stored(ad,DB)

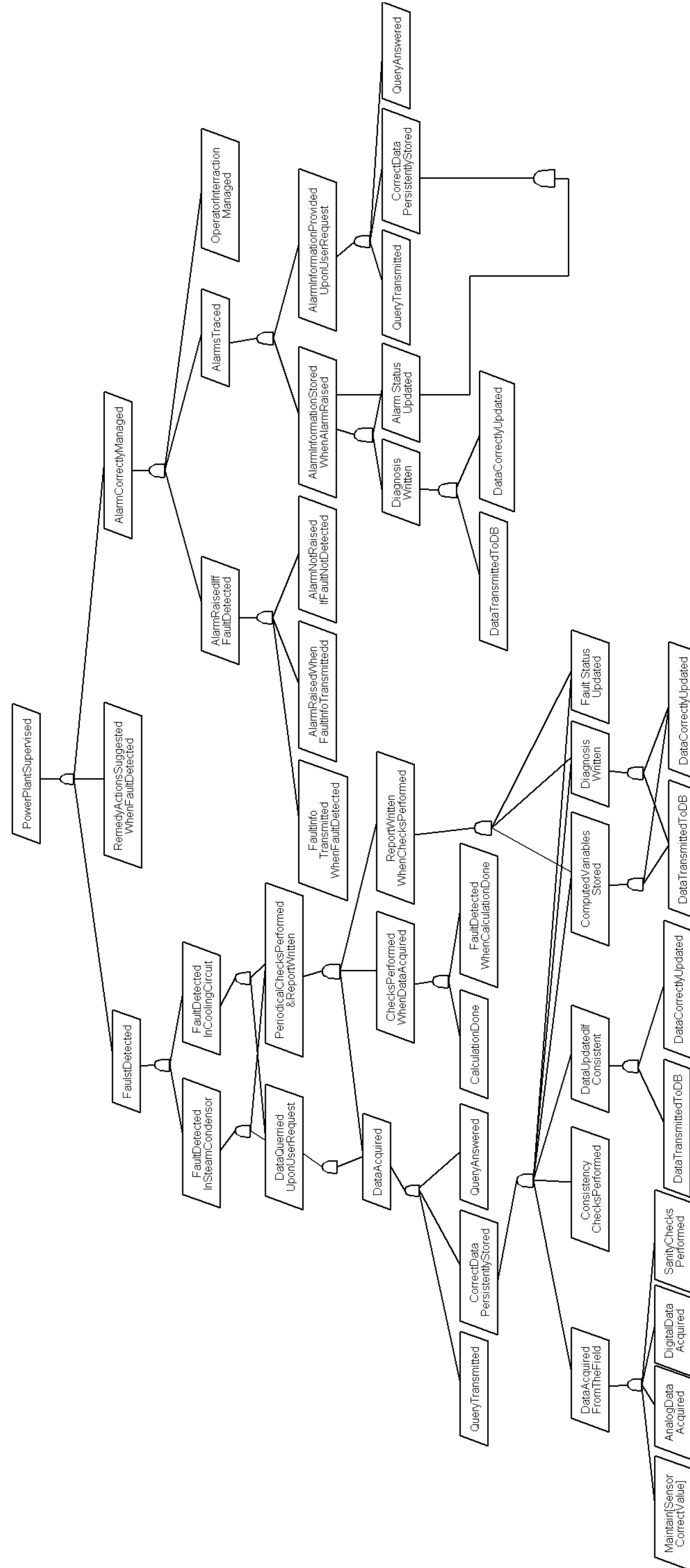

Figure A.1: Functional goal diagram

3. **Goal** `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) $\Rightarrow \Diamond$ Answer(ai) $\wedge$ Representation(ai,a)

4. **Goal** `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) $\wedge$ Representation(ai,a) $\Rightarrow \Diamond$ Stored(ai,DB)
    $\wedge$ Stored(fd,DB) $\wedge$ Concerns(fd,fi,ai)
    $\wedge$ PowerPlant.AlarmStatus ='on'

5. **Goal** `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) $\Rightarrow \blacklozenge$ Detected(f,l)

6. **Goal** `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)
$\bigwedge$ $\forall$ a:Alarm, $\exists$!f:Fault,$\exists$!l:Location
Raise(f,a) $\Rightarrow \blacklozenge$ Detected(f,l)

7. **Goal** `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)

8. **Goal** `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'

9. **Goal** `AlarmTraced`

   Def Informations on alarms previously raised can be retrieved

   Concerns Alarm

   AndRefines AlarmCorrectlyManaged

   RefinedTo AlarmInformationStoredWhenAlarmRaised, AlarmInformationProvidedUponUserRequest

10. **Goal** `AnalogDataAcquired`

    Def All the data coming from working analog sensors are acquired

    Concerns Sensor, SensorInformation

    AndRefines DataAcquiredFromTheField

    FormalDef $\forall$ s: Sensor, si:SensorInformation
    s.type ='Analog' $\wedge$ s.status ='on' $\wedge$ s.id = si.id $\Rightarrow \Diamond$ Acquired(s) $\wedge$ s.DataValue=si.DataValue

11. **Goal** `CalculationDone`

    Def All the calculations needed to detect fault in the Power-Plant are done

    Concerns SensorInformation

    AndRefines ChecksPerformedWhenDataAcquired

    UnderResponsabilityOf PRECON

    FormalDef $\forall$ si:SensorInformation
    Transmitted(si,DB,PRECON) $\Rightarrow \Diamond$ CalculationDone

12. **Goal** `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, FaultDetectedWhenCalculation-Done

    FormalDef $\forall$ f:Fault, si: SensorInformation, l:Location
    Occurs(f,l) $\wedge$ Transmitted(si,DB,PRECON)
    $\Rightarrow \Diamond_{\leq 5min}$ Detected(f,l)
    $\bigwedge \neg$ Occurs(f,l) $\wedge$ Transmitted(si,DB,PRECON)
    $\Rightarrow \Diamond \neg$ Detected(f,l)

13. **Goal** `ComputedVariablesStored`

    Def The variables resulting from the different calculations need in order to detect Fault must be stored.

    AndRefines ReportWrittenWhenCheckPerformed

    RefinedTo DataTransmittedToDB

14. **Goal** `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)

15. **Goal** `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)

16. **Goal** `DataAcquired`

    Def All the data needed are acquired from the field

    Concerns Sensor, SensorInformation

    AndRefines DataQuerriedUponUserRequest, PeriodicalChecksPerformed

    RefinedTo CorrectDataPersistentlyStored, QuerryTransmitted, QuerryAnswered

    FormalDef $\forall$ s:Sensor, $\exists$! si: SensorInformation Querry(s) $\Rightarrow \Diamond_{\leq 2s}$ Transmitted(si,DB,PRECON) $\wedge$ Representation(si,s)

17. **Goal** `DataAcquiredFromTheField`

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

    Concerns Sensor, SensorInformation

    AndRefines CorrectDataPersistentlyStored

    RefinedTo AnalogDataAcquired, DigitalDataAcquired, SanityCheckPerformed

    FormalDef $\forall$ s: Sensor, si:SensorInformation s.type = 'Digital' $\vee$ s.type ='Analog' $\wedge$ s.id=si.id $\Rightarrow \Diamond$ Acquired(s) $\wedge$ s.DataValue= si.DataValue

18. **Goal** `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)

19. **Goal** `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)

20. **Goal** `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) $\wedge$ Representation(si,s)

21. **Goal** `DataTransmittedToDB`

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

    Concerns Alarm, AlarmInformation, FaultInformation

    AndRefines AlarmInformationStoredWhenAlarmRaised

    UnderResponsabilityOf COMMUNICATION

FormalDef $\forall$ a:Alarm, $\exists$! fi: FaultInformation,
$\exists$! ai:AlarmInformation, $\exists$!ad: AlarmDiagnosis
Raise(fi,a) $\wedge$ Representation(ai,a)
$\Rightarrow \Diamond$ Transmitted(ai,ALARM,DB) $\wedge$ Transmitted(ad,ALARM,DB)
$\wedge$ Concerns(ad,fi,ai)

22. **Goal** `DataTransmittedToDB`

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

Concerns Fault , FaultInformation, SensorInformation

AndRefines FaultDiagnosisWritten, ComputedVariablesStored

UnderResponsabilityOf COMMUNICATION

FormalDef $\forall$ f:Fault, $\exists$ l:Location, $\exists$! fi: FaultInformation,
$\exists$! si: SensorInformation $\exists$!ad: FaultDiagnosis
Detected(f,l) $\wedge$ Representation(fi,f)
$\Rightarrow \Diamond$ Transmitted(fi,PRECON,DB) $\wedge$ Transmitted(fd,ALARM,DB)
$\wedge$ Concerns(ad,si,fi)

23. **Goal** `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)
$\wedge$ Representation(si,s)

24. **Goal** `DigitalDataAcquired`

Def All the data coming from working digital sensors are acquired

Concerns Sensor, SensorInformation

AndRefines DataAcquiredFromTheField

UnderResponsabilityOf Acquisition Unit

FormalDef $\forall$ s: Sensor, si:SensorInformation
s.type ='Digital' $\wedge$ s.status ='on' $\wedge$ s.id=si.id $\Rightarrow \Diamond$ Acquired(s)
$\wedge$ s.DataValue=si.DataValue

25. **Goal** `FaultDetected`

Def Faults occurring in any location (i.e.,the steam condenser or the cooling) circuit must be detected. Fault occur in only one location at a time.

Concerns SteamCondensor, CoolingCircuit, Fault

AndRefines PerformanceOfThePlantMonitored

RefinedTo FaultDetectedInSteamCondensor, FaultDetectedInCoolingCircuit

FormalDef $\forall$ f:Fault, $\exists$!l:Location
Occurs(f,l) $\Rightarrow \Diamond$ Detected(f,l)

26. **Goal** `FaultDetectedInCoolingCircuit`

Def Faults in the cooling circuit must be detected

Concerns CoolingCircuit, Fault

AndRefines FaultDetected

RefinedTo DataQuerriedUpondUserRequest, PeriodicalChecksPerformed&RepportsWritten

FormalDef $\forall$ f:Fault
Occurs(f,CoolingCircuit) $\Rightarrow \Diamond$ Detected(f,CoolingCircuit)

27. **Goal** `FaultDetectedInSteamCondenser`

Def Faults in the steam condenser must be detected

Concerns SteamCondenser, Fault

AndRefines FaultDetected

RefinedTo DataQuerriedUpondUserRequest, PeriodicalChecksPerformed&RepportsWritten

FormalDef $\forall$ f:Fault
Occurs(f,SteamCondenser) $\Rightarrow \Diamond$ Detected(f,SteamCondenser)

28. **Goal** `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)
$\bigwedge$ CalculationDone $\wedge \neg$ Occurs(f,l) $\Rightarrow \Box \neg$ Detected(f,l)

29. **Goal** `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) $\Rightarrow \lozenge$ Store(fi,DB) $\wedge$ Representation(fi,f)
    $\wedge$ Stored(fd,DB) $\wedge$ Concerns(ds, di,si)

30. **Goal** `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) $\wedge$ Representation(fi,f)
    $\Rightarrow \lozenge$ Transmitted(fi, PRECON, ALARM)

31. **Goal** `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'

32. **Goal** `Maintain[SensorCorrectValue]`

    Def The values of the variables measured by the sensor should be equal to the actual values of the variables.

    Concerns Sensor

    AndRefines DataAcquiredFromTheField

    UnderResponsabilityOfSensor Sensor

FormaDef $\forall$ s:Sensor, l:Location
Monitors(s,l) $\wedge$ s.DataType=Temperature
$\Rightarrow \Box$ s.DataValue=l.Temperature
$\bigwedge$ Monitors(s,l) $\wedge$ s.DataType=Pressure
$\Rightarrow \Box$ s.DataValue=l.Pressure

33. **Goal** `PowerPlantSupervised`

Def The system must continuously supervise the power plant in order to detect faults in the steam condenser or in the cooling circuit and to raise appropriate alarms in case of fault detection. Moreover, it supports the operators suggesting remedy actions.

Concerns PowerPlant, SteamCondensor, CoolingCircuit

RefinedTo FaultDetected, RemedyActionSuggestedWhenFault-Detected, AlarmCorrectlyManaged

34. **Goal** `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, FaultDetected-InCoolingCircuit

RefinedTo DataAcquired, ChecksPerformedWhenDataAcquired, ReportWrittenWhenChecksPerformed

FormalDef $\forall$ f: Fault, $\exists$! l:Location
Occurs(f,l) $\Rightarrow \Diamond_{\leq 5min}$ Detected(f,l)

35. **Goal** `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)

36. **Goal** `RemedyActionsSuggestedWhenFaultDetected`

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

Concerns SteamCondensor, CoolingCircuit, Fault
AndRefines PerformanceOfThePlantMonitored
UnderResponsabilityOf PRECON

37. **Goal** `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 ∀ f: Fault, ∃ !fi: FaultInformation, ∃ !l:Location, ∃ ! fd: FaultDiagnosis,∃ !si:SensorInformation
    Detected(f,l) ⇒ ◊ Stored(fi,DB) ∧ Representation(fi,f)
    ∧ Stored(fd,DB) ∧ Concerns(fd,si,fi)

38. **Goal** `SanityCheckPerformed`

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

    Concerns Sensor

    AndRefines DataAcuiredFromTheField

    FormalDef ∀ s:Sensor
    s.workingProperly = false ∧ s.status = 'on'
    ⇒ ∘ s.status = 'off'
    ⋀ s.workingProperly = true ∧ s.status ='off'
    ⇒ ∘ s.status = 'on'

A.1.2 Non-functional goals

Non-functional goals are listed by category. Figure A.2 shows the NFG diagram.

- *Reliability goals*

  Goal `NoDataIntroduced`

  FormalDef ∀ x:Data, X:$\mathcal{P}$(Data)[1], A_1,A_2:Agent
  Transmitted(X,A_1,A_2) ∧ x ∈ Transmitted(X,A_1,A_2) ⇒ x ∈ X

[1] $\mathcal{P}$(Data) denotes the set of subsets of Data

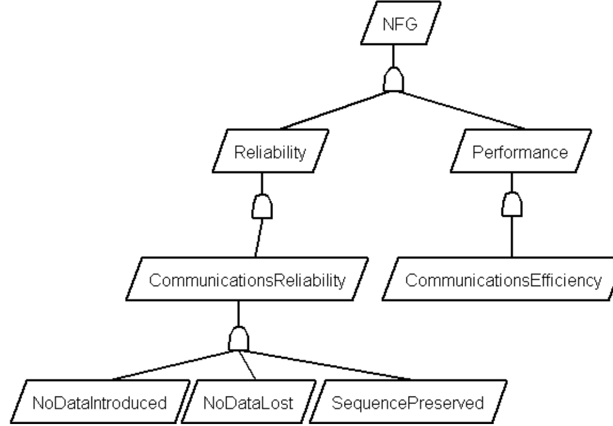

Figure A.2: Non-functional goal diagram

Goal `NoDataLost`

FormalDef $\forall$ x:Data, X:$\mathcal{P}$(Data),A_1,A_2:Agent
$x \in X \wedge \text{Transmitted}(X,A_1,A_2) \Rightarrow x \in \text{Transmitted}(X,A_1,A_2)$

Goal `SequencePreserved`

FormalDef $\forall$ x,y: Data, X:$\mathcal{P}$(Data), A_1,A_2:Agent, $\exists$ u,v: Data
$x,y \in X \wedge \text{Before}(x,y,X) \wedge \text{Transmitted}(X,A_1,A_2)$
$\Rightarrow u,v \in \text{Transmitted}(X,A_1,A_2) \wedge \text{Before}(u,v,\text{Transmitted}(X,A_1,A_2))$
$\wedge\ x = u \wedge y = v$

- *Performance goals*

Goal `CommunicationEfficiency`[2]

FormalDef $\neg$ Transmitted(fi,PRECON,ALARM) $\Rightarrow \Diamond_{\leq 1s}$ Transmitted(fi,PRECON,ALARM)

A.2 Object Specifications

The objects are listed in alphabetical order. Figure A.3 shows the object diagram.

1. **Entity** `Alarm`

[2]The goal is actually the instantiation of a more general one in the particular case of the transmission of a FaultInformation between PRECON and ALARM

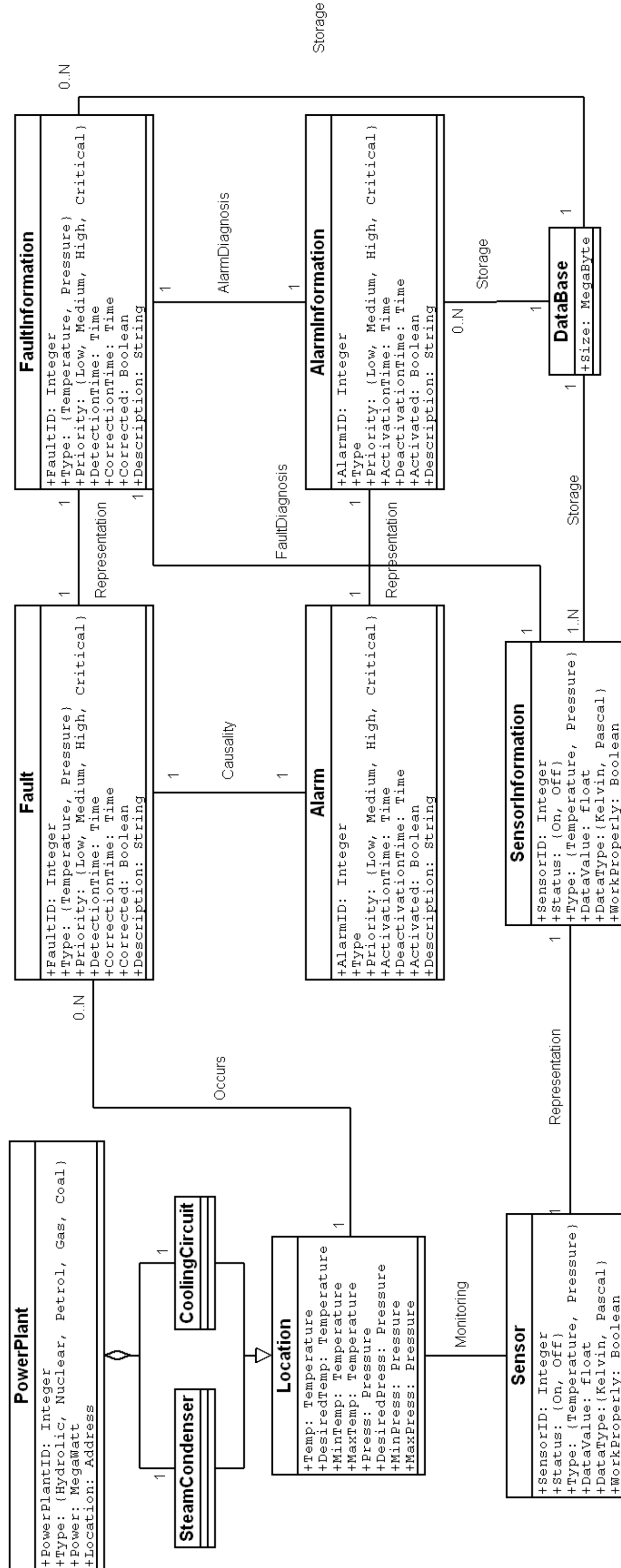

Figure A.3: ObjectDiagram

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 ⇒ DeactivationTime = null
Activated = false ⇒ DeactivationTime ≠ null

DomInit Activated = true
DeactivationTime = null

2. **Entity** `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

3. **Entity** `CoolingCircuit`

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.

IsA Location

Has **Inherited from** Location
Temp: Temperature
DesiredTemp: Temperature
MinTemp: Temperature

MaxTemp: Temperature
Press: Pressure
DesiredPress: Pressure
MinPress: Pressure
MaxPress: Pressure

DomInvar MinTemp $\leq$ Maxtemp
MinPress $\leq$ MaxPress

DomInit /

4. **Entity** `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

5. **Entity** `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

6. **Entity** `Location`

Def represent a location in the power plant
Has Temp: Temperature
DesiredTemp: Temperature
MinTemp: Temperature
MaxTemp: Temperature
Press: Pressure
DesiredPress: Pressure
MinPress: Pressure
MaxPress: Pressure
DomInvar MinTemp $\leq$ Maxtemp
MinPress $\leq$ MaxPress
DomInit /

7. **Entity** `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

8. **Entity** `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

9. **Entity** `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

10. **Entity** `SteamCondenser`

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

IsA Location

Has **Inherited from** Location
Temp: Temperature
DesiredTemp: Temperature

MinTemp: Temperature
MaxTemp: Temperature
Press: Pressure
DesiredPress: Pressure
MinPress: Pressure
MaxPress: Pressure

DomInvar MinTemp $\leq$ Maxtemp
MinPress $\leq$ MaxPress

DomInit /

A.3 Agents Specifications

The agents are listed by alphabetical order. Figure A.4 shows the responsibility assignment of the different agents and Figure A.5 shows the agents interfaces in terms of its monitored and controlled variables.

The agents are listed by alphabetical order. Figure A.4 shows the responsibility assignment of the different agents.

1. **Agent** `Acquisition Unit`

   Def An agent that acquires data from the various sensors monitoring the power plant

   Has /

   Monitors Sensor/ID, Sensor/Status, Sensor/Type, Sensor/DataValue, Sensor/DataType

   Controls SensorInformation/ID, SensorInformation/Status, SensorInformation/Type, SensorInformation/DataValue, SensorInformation/DataType

   ResponsibleFor AnalogDataAcquired, DigitalDataAcquired

   DependsOn Sensor

   Perfoms Acquire Analog Data, Acquire Digital Data

2. **Agent** `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

Figure A.4: Responsibility assignment of the different agents

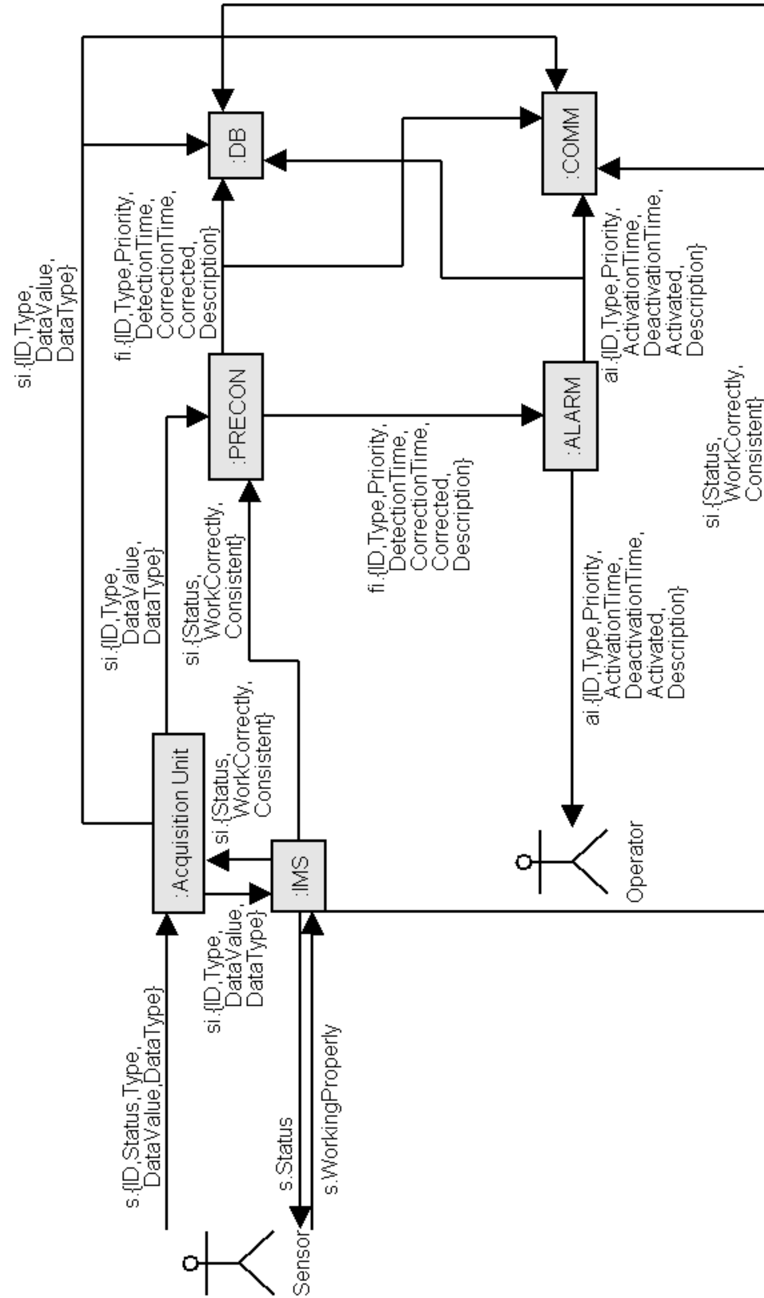

Figure A.5: Agent context diagram

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

3. **Agent** `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

4. **Agent** `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

5. **Agent** `Instrumentation Maintenance System`(IMS)

   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

6. **Agent** `OPERATOR`

   Def Represents user who interacts with the system

   Has /

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

   Controls /

   ResponsibleFor OperatorInteractionsManaged

   DependsOn /

   Perfoms Manages Operator Interaction

7. **Agent** `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

8. **Agent** `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, Sensor/Status,

Controls Sensor/SensorID, Sensor/Type, Sensor/DataValue, Sensor/DataType

ResponsibleFor Maintain[SensorCorrectValue]

A.4 Operations specifications

Operations are listed by alphabetical order.

1. **Operation** `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 Acquisistion Unit

2. **Operation** `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 Acquisition Unit

3. **Operation** `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

4. **Operation** `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

5. **Operation** `Calculate`

   Def calculate all needed things in order to detect faults

   Input si: SensorInformation

   Output /

   DomPre ¬ CalculationDone

   DomPost CalculationDone

   ReqTrig For CalculationDone
   ¬ CalculationDone $\mathbf{S}_{=1s}$ Transmitted(si,DB,PRECON) ∧ ¬ CalculationDone

   PerformedBy PRECON

6. **Operation** `DetectFault`

   Def detect Fault

   Input f: Fault, l: Location

   Output /

   DomPre ¬ Detected(f,l)

   DomPost Detected(f,l)

   ReqTrig For FaultDetectedWhenCalculationDone
   ¬ Detected(f,l) $\mathbf{S}_{=1s}$ CalculationDone ∧ Occurs(f,l) ∧ ¬ Detected(f,l)

   PerformedBy PRECON

7. **Operation** `RaiseAlarm`

   Def Raise the alarm

   Input fi: FaultInformation, a: Alarm

   Output a: Alarm

   DomPre ¬ Raise(fi,a)

   DomPost Raise(fi,a)

   ReqTrig For AlarmRaisedWhenFaultInformationTransmitted
   ¬ Raise(fi,a) $\mathbf{S}_{=1s}$ Transmitted(fi,PRECON, ALARM) ∧ ¬ Raise(fi,a)

   PerformedBy ALARM

8. **Operation** `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

9. **Operation** `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

10. **Operation** `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

11. **Operation** `SwitchFaultStatusOn`

 Def switch the Fault Status on
 Input f: Fault, l: Location, PowerPlant
 Output PowerPlant/FaultStatus
 DomPre PowerPlant.FaultStatus = off
 DomPost PowerPlant.FaultStatus =on
 ReqTrig For FaultStatusUpdated
  Detected(f,l)
 PerformedBy PRECON

12. **Operation** `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

13. **Operation** `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

14. **Operation** `TransmitAlarmData`

    Def Transmit the alarm data to the DataBase

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

    Output /

    DomPre ¬ Transmitted(ai,ALARM,DB) ∨ ¬ Transmitted(ad,ALARM,DB)
    ∨ ¬ Concerns(ad,fi,ai)

    DomPost Transmitted(ai,ALARM,DB) ∧ Transmitted(ad,ALARM,DB)
    ∧ Concerns(ad,fi,ai)

    ReqTrig For AlarmDataTransmitted
    ¬ Transmitted(ai,ALARM,DB) ∨ ¬ Transmitted(ad,ALARM,DB)
    ∨ ¬ Concerns(ad,fi,ai) $\mathbf{S}_{=1s}$ Raise(fi,a) ∧ a.ID = ai.ID ∧ $\Big($ ¬
    Transmitted(ai,ALARM,DB) ∨ ¬ Transmitted(ad,ALARM,DB)
    ∨ ¬ Concerns(ad,fi,ai) $\Big)$

    PerformedBy COMMUNICATION

15. **Operation** `TransmitAlarmQuery`

    Def transmit a alarm query to the DataBase

    Input a: Alarm

    Output /

    DomPre $\neg$ 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

16. **Operation** `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

17. **Operation** `TransmitFaultInformation`

    Def Transmit Fault Information to The ALARM Management unit

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

    Output /

    DomPre $\neg$ Transmitted(fi,PRECON, ALARM)

    DomPost Transmitted(fi,PRECON, ALARM)

    ReqTrig For FaultInformationTransmittedWhenFaultDetected
    $\neg$ Transmitted(fi,PRECON, ALARM) $\mathbf{S}_{=1s}$ Detected(f,l) $\wedge$ f.ID = fi.ID $\wedge$ $\neg$ Transmitted(fi,PRECON, ALARM)

PerformedBy COMMUNICATION

18. **Operation** `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

19. **Operation** `TransmitSensorQuery`

    Def transmit a sensor query to the DataBase

    Input s: Sensor

    Output /

    DomPre ¬ Transmitted(s,PRECON,DB)

    DomPost Transmitted(s,PRECON,DB)

    ReqTrig For SensorQuerryTransmitted
    ¬ Transmitted(s,PRECON,DB) $\mathbf{S}_{=1s}$ Query(s) ∧ ¬ Transmitted(s,PRECON,DB)

    PerformedBy COMMUNICATION

20. **Operation** `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

21. **Operation** `UpdateAlarmData`

    Def Update Alarm data in the DataBase

    Input ai: AlarmInformation, ad: AlarmDiagnosis

    Output /

    DomPre $\neg$ Stored(ai) $\vee$ $\neg$ Stored(ad)

    DomPost Store(ai) $\wedge$ Stored(ad)

    ReqTrig For AlarmDataCorrectlyUpdated
    $\neg$ Stored(ai) $\vee$ $\neg$ Stored(ad) $\mathbf{S}_{=1s}$ Transmitted(ai,ALARM,DB)
    $\wedge$ Transmitted(ad,ALARM,DB) $\wedge$ ($\neg$ Stored(ai) $\vee$ $\neg$ Stored(ad)
)

    PerformedBy

22. **Operation** `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) $\wedge$ ($\neg$ Stored(fi) $\vee$ $\neg$ Stored(fd))

    PerformedBy DB

23. **Operation** `UpdateSensorData`

    Def Update the data in the DataBase

    Input si: SensorInformation

    Output /

    DomPre $\neg$ Stored(si)

    DomPost Stored(si)

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

    PerformedBy DB

24. **Operation** `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
(si.DataType = 'Temperature' ∧ (minTemp ≤ si.Value ≤ maxTemp))
∨ (si.DataType = 'Pressure' ∧ minPres ≤ si.Value ≤ maxPres))

PerformedBy ACQUISITION UNIT

A.5 Modifications resulting from the obstacle analysis

Figure A.6 shows the modifications brought to the NFG diagram. Modifications are highlighted through thicker lines. The following goals have been added:

- *Safety goals*

Goal `Avoid[AllSensorOff]`

Def There must always be at least one sensor working

Concerns Sensor

FormalDef □ ¬ ∀ s:Sensor
s.status=off

Goal `Avoid[LocationNotMonitored]`

Def Every location must always be monitored by at least a sensor

Concerns Sensor

FormalDef □ ¬ ∃ l:Location, ∀ s:Sensor
¬ Monitor(s,l)

- *Reliability goals*

Goal `Maintain[FaultTolerantCommunication(PRECON,ALARM)]`

Def The communication between PRECON and ALARM should be fault-tolerant

- *Accuracy Goals*

Goal `Maintain[AccurateData(Acquisition Unit,IMS)]`

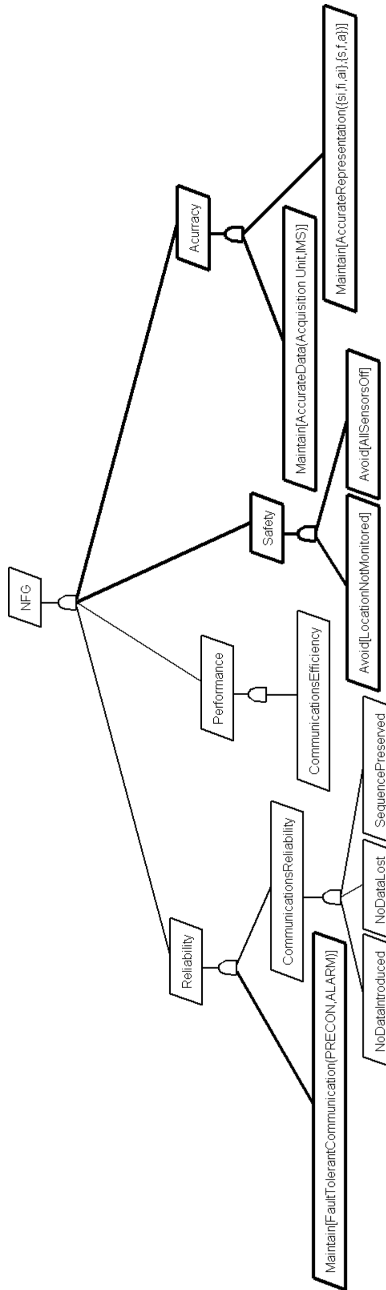

Figure A.6: Modifications brought to the NFG diagram after the obstacle analysis

Def The Sensor informations possessed by the agents Acquisition Unit an IMS should always be equal

Concerns SensorInformation

Goal `Maintain[AccurateRepresentation(ai,a)]`

Def The software representation of an alarm should always reflect the alarm state

Concerns Alarm, AlarmInformation

FormalDef ∀ a:Alarm, ∃! ai:AlarmInformation
a.id=ai.id ⇒ □ (a.Type = ai.Type ∧ a.Priority = ai.Priority
∧ a.ActivationTime = ai.ActivationTime
∧ a.CorfectionTime = ai.CorrectionTime
∧ a.Corrected = ai.Corrected
∧ a.Description = ai.Description)

Goal `Maintain[AccurateRepresentation(fi,f)]`

Def The software representation of a Fault should always reflect the fault state

Concerns Fault, FaultInformation

FormalDef ∀ f:Fault, ∃! fi:FaultInformation
f.id=fi.id ⇒ □ (f.Type = fi.Type ∧ f.Priority = fi.Priority
∧f.DetectionTime = fi.DetectionTime
∧ f.CorfectionTime = fi.CorrectionTime
∧ f.Corrected = fi.Corrected
∧ f.Description = fi.Description)

Goal `Maintain[AccurateRepresentation(si,s)]`

Def The software representation of a sensor should always reflect the sensor state

Concerns Fault, FaultInformation

FormalDef ∀ s:Sensor, ∃! si:SensorInformation
s.id=si.id ⇒ □ (s.Status=si.Status ∧ s.Type = si.Type
∧ s.DataValue = si.DataValue ∧ s.DataType = si.DataType
∧ s.WorkingCorrectly = si.WorkingCorrectly)

Appendix B

Architectural Prescriptions

B.1 Initial Prescriptions

Preskriptor Specification: PowerPlant Monitoring System

Problem Goals Specifications: PowerPlant Monitoring Process

Components:

1. **Component** `PowerPlantSupervisingSystem`[1,1]
   Type Processing
   Constraints PerformancOfThePlantMonitored
   Composed of PRECON
   ALARM
   DataBase
   Communication
   Uses /
2. **Component** `PRECON`[1,1]
   Type Processing
   Constraints FaultDetected
   RemedyActionSuggested
   PeriodicalChecksPerformed&ReportWritten
   Composed of FaultDetectionEngine
   FaultInformation
   FaultDiagnosis

SensorInformation
SensorConnect

Uses /

3. **Component** `ALARM`[1,1]

    Type Processing

    Constraints AlarmCorrectlyManaged
    AlarmRaisedIffFaultDetected
    AlarmTraced

    Composed of AlarmManager
    AlarmInformation
    AlarmDiagnosis
    InteractionManager

    Uses /

4. **Component** `Database`[1,1]

    Type Processing

    Constraints CorrectDataPersistentlyStored

    Composed of QueryManager
    UpdateManager

    Uses /

5. **Component** `Communication`[1,1]

    Type Connector

    Constraints NoDataIntroduced
    NoDataLost
    SequencePreserved
    DataTransmittedInTime
    DataTransmittedToTheDB
    QueryTransmitted
    FaultInformationTransmittedWhenFaultDetected

    Composed of UpdateDBConnect
    QueryDBConnect
    FaultDetectionEngineAlarmManagerConnect

    Uses /

6. **Component** `FaultDetectionEngine`[1,1]

    Type Processing

    Constraints CalculationDone
    FaultDetectedWhenCalculationDone
    FaultStatusUpdated

CheckPerformedWhenDataAcquired
ReportWrittenWhenCheckPerformed

Composed of /

Uses SensorConnect *to interract with* SensorInformation
FaultDetectionEngineAlarmManagerConnect *to interract with* AlarmManager
UpdateDBConnect *to interact with* UpdateManager

7. **Component** `FaultInformation`[0,n]

   Type Data

   Constraints FaultInformationTransmittedWhenFaultDetected

   Composed of /

   Uses FaultDetectionEngineAlarmManagerConnect *to interract with* AlarmManager
   UpdateDBConnect *to interact with* UpdateManager

8. **Component** `FaultDiagnosis`[0,n]

   Type Data

   Constraints DiagnosisWritten
   ComputedVariablesStored

   Composed of /

   Uses UpdateDBConnect *to interact with* DBUpdateManager

9. **Component** `SensorInformation`[l,n][1]

   Type Data

   Constraints AnalogDataAcquired
   DigitalDataAcquired
   SanityCheckPerformed
   ConsistencyCheck

   Composed of /

   Uses UpdateDBConnect *to interact with* DB
   SensorConnect *to interact with* FaultDetectionEngine

10. **Component** `SensorConnect`[1,1]

    Type Connector

    Constraints DataAcquiredFromTheField

    Composed of /

[1] To achieve the non-functional goal `Avoid[LocationNotMonitored]` there must be at least l sensors where l is the number of locations

Uses /

11. **Component** `UpdateDBConnect`[1,1]

    Type Connector

    Constraints Secure
    TimeConstraint = 2s

    Composed of /

    Uses /

12. **Component** `QueryDBConnect`[1,1]

    Type Connector

    Constraints TimeConstraint = 5s

    Composed of /

    Uses /

13. **Component** `FaultDetectionEngineAlarmManagerConnect`[1,1]

    Type Connector

    Constraints Maintain[FaultTolerantCommunication(PRECON,ALARM)]
    Secure
    TimeConstraint = 1s

    Composed of /

    Uses /

14. **Component** `AlarmManager`[1,1]

    Type Processing

    Constraints AlarmRaisedWhenFaultInformationTransmitted
    FaultInformationTransmitted
    AlarmStatusUpdated
    AlarmNotRaisedIfFaultNotDetected

    Composed of /

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

15. **Component** `AlarmInformation`[0,n]

    Type Data

    Constraints AlarmInformationStoredWhenAlarmRaised

    Composed of /

    Uses UpdateDBConnect *to interact with* UpdateManager

16. **Component** `AlarmDiagnosis`[0,n]

Type Data
Constraints DiagnosisWritten
Composed of /
Uses UpdateDBConnect *to interact with* UpdateManager

17. **Component** `InteractionManager`[1,1]
    Type Processing
    Constraints OperatorInteractionManaged
    Composed of /
    Uses QueryDBConnect *to interact with* QueryManager
18. **Component** `QueryManager`[1,1]
    Type Processing
    Constraints QueryAnswered
    DataQueriedUponUserRequest
    AlarmInformationProvidedUponUserRequest
    DataAcquired
    Composed of /
    Uses QueryDBConnect *to interact with* InteractionManager
19. **Component** `UpdateManager`[1,1]
    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

B.2 Prescriptions resulting from step 4

Definition of the new constraints

1. **Constraint** `Achieve[SwitchInCaseOfFailure]`
   Informal Def : Every time a component fails (PRECON or ALARM), the copy should take te relay

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

2. **Constraint** `Maintain[OneActiveComponent]`

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

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

3. **Constraint** `Maintain[NoPrecedenceRelation]`

   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

4. **Constraint** `Achieve[TransparencyOfSwitching]`

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

Prescriptions of the modified components

The modifications are highlighted in bold.

1. **Component** `FaultDetectionEngine`[**2,2**]

   Type Processing

   Constraints CalculationDone
   FaultDetectedWhenCalculationDone
   FaultStatusUpdated
   CheckPerformedWhenDataAcquired
   ReportWrittenWhenCheckPerformed

   Composed of /

   Uses SensorConnect *to interract with* SensorInformation
   FaultDetectionEngineAlarmManagerConnect *to interract with* AlarmManager
   UpdateDBConnect *to interact with* UpdateManager

2. **Component** `AlarmManager`[**2,2**]

    Type Processing

    Constraints AlarmRaisedWhenFaultInformationTransmitted
    FaultInformationTransmitted
    AlarmStatusUpdated
    AlarmNotRaisedIfFaultNotDetected

    Composed of /

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

3. **Component** `FaultDetectionEngineAlarmManagerConnect`[1,1]

    Type Connector

    Constraints Maintain[FaultTolerantCommunication(PRECON,ALARM)]
    Secure
    TimeConstraint = 1s
    Achieve[**SwitchInCaseOfFailure**]
    Maintain[**OneActiveComponent**]
    Maintain[**NoPrecedenceRelation**]
    Achieve[**TransparencyOfSwitching**]

    Composed of /

    Uses /

Appendix C

Wright Specifications

C.1 The Fault-Tolerant Communication Pattern

C.1.1 Initial Wright Specification

component PRECON

port AcquisitionInput = receive?si → AcquisitionInput

port ALARMOutput = $\overline{ALARMOutput.Detected(f,l)}$

computation = Calculate [] SwitchFaultStatusOff **where**

Calculate = AcquisitionInput.receive?si → calculationDone

$$\text{DetecFault} = \begin{cases} \overline{ALARMOutput.Detected(f,l)} & \\ \rightarrow SwitchFaultStatusOn & \textbf{when } Occurs(f,l) \\ \textbf{computation} & \textbf{when} \neg Occurs(f,l) \end{cases}$$

$$\text{SwitchFaultStatusOn} = \begin{cases} FaultStatusOn & \\ \rightarrow \textbf{computation} & \textbf{when } FaultStatusOff \\ \textbf{computation} & \textbf{when } FaultStatusOn \end{cases}$$

$$\text{SwitchFaultStatusOff} = \begin{cases} FaultStatusOff & \\ \rightarrow \textbf{computation} & \textbf{when } \; FaultStatusOn \\ & \qquad \wedge \neg Occurs(f) \end{cases}$$

component ALARM

port PRECONInput = Detected(f,l) → $\overline{request!f}$ → receive?fi → PRECONInput

computation = RaiseAlarm [] SwitchAlarmStatusOff **where**

RaiseAlarm = PRECONInput.receive?fi → Raise(fi,a) → SwitchAlarm-StatusOff

$$\text{SwitchAlarmStatusOn} = \begin{cases} AlarmStatusOn & \\ \rightarrow \textbf{computation} & \textbf{when } AlarmStatusOff \\ \textbf{computation} & \textbf{when } AlarmStatusOn \end{cases}$$

$$\text{SwitchAlarmStatusOff} = \begin{cases} AlarmStatusOff & \\ \rightarrow \textbf{computation} & \textbf{when} \quad AlarmStatusOn \\ & \quad \wedge \neg Raise(fi, a) \end{cases}$$

constraints ∀ fi:FaultInformation, ∃ !a:Alarm
Transmitted(fi,PRECON,ALARM) ⇒ $\Diamond_{\leq 1s}$ Raise(fi,a)

connector Pull-Dataflow

role Producer = $\overline{dataReady}$ → Producer
[] request?x → $\overline{send!x}$ → Producer ⊓ §

role Consumer = dataReady → $\overline{request!x}$ → receive?x → Consumer
[] §

glue = Producer.dataReady → $\overline{Consumer.dataReady}$ → Consumer.request!x
→ $\overline{Producer.request?x}$ → Producer.send!x → $\overline{Consumer.reveive?x}$
→ **glue** [] §

instances Precon: PRECON
Alarm: ALARM
Precon2Alarm: PullDataflow

attachements Precon.ALARMOutput **as** Precon2Alarm.Producer
Alarm.PRECONInput **as** Precon2Alarm.Consumer

C.1.2 Resulting Wright Specification

component PRECON

port AcquisitionInput = receive?si → AcquisitionInput

port ALARMOutput = $\overline{ALARMOutput.Detected(f, l)}$

$$\textbf{port} \text{ Copylink=} \begin{cases} isAlive \rightarrow \overline{ImAlive} \rightarrow \textbf{Copylink} \\ [] \ reset \rightarrow sleep \rightarrow wakeUp \rightarrow \textbf{Copylink} \end{cases}$$

$$\textbf{computation} = \begin{cases} compute \\ [] \ reset \rightarrow sleep \rightarrow wakeUp \rightarrow \textbf{computation} \\ [] \ Copylink.isAlive \rightarrow \overline{Copylink.ImAlive} \rightarrow \textbf{computation where} \end{cases}$$

compute = Calculate [] SwitchFaultStatusOff
Calculate = AcquisitionInput.receive?si → calculationDone

$$\text{DetecFault} = \begin{cases} \overline{ALARMOutput.Detected(f,l)} & \\ \rightarrow SwitchFaultStatusOn & \textbf{when } Occurs(f,l) \\ \textbf{computation} & \textbf{when} \neg Occurs(f,l) \end{cases}$$

$$\text{SwitchFaultStatusOn} = \begin{cases} FaultStatusOn & \\ \rightarrow \textbf{computation} & \textbf{when } FaultStatusOff \\ \textbf{computation} & \textbf{when } FaultStatusOn \end{cases}$$

$$\text{SwitchFaultStatusOff} = \begin{cases} FaultStatusOff & \\ \rightarrow \textbf{computation} & \textbf{when } \quad FaultStatusOn \\ & \qquad \wedge \neg Occurs(f) \end{cases}$$

component ALARM

port PRECONInput = Detected(f,l) $\rightarrow \overline{request!f} \rightarrow$ receive?fi $\rightarrow$ PRECONInput

$$\textbf{port} \quad \text{Copylink} = \begin{cases} isAlive \rightarrow \overline{ImAlive} \rightarrow \textbf{Copylink} \\ [] \; reset \rightarrow sleep \rightarrow wakeUp \rightarrow \textbf{Copylink} \end{cases}$$

$$\textbf{computation} \quad = \begin{cases} compute \\ [] \; reset \rightarrow sleep \rightarrow wakeUp \rightarrow \textbf{computation} \\ [] \; Copylink.isAlive \rightarrow \overline{Copylink.ImAlive} \rightarrow \textbf{computation where} \end{cases}$$

compute= RaiseAlarm [] SwitchAlarmStatusOff
RaiseAlarm = PRECONInput.receive?fi $\rightarrow$ Raise(fi,a) $\rightarrow$ SwitchAlarm-StatusOff

$$\text{SwitchAlarmStatusOn} = \begin{cases} AlarmStatusOn & \\ \rightarrow \textbf{computation} & \textbf{when } AlarmStatusOff \\ \textbf{computation} & \textbf{when } AlarmStatusOn \end{cases}$$

$$\text{SwitchAlarmStatusOff} = \begin{cases} AlarmStatusOff & \\ \rightarrow \textbf{computation} & \textbf{when } \quad AlarmStatusOn \\ & \qquad \wedge \neg Raise(fi,a) \end{cases}$$

constraints $\forall$ fi:FaultInformation, $\exists$!a:Alarm
Transmitted(fi,PRECON,ALARM) $\Rightarrow \Diamond_{\leq 1s}$ Raise(fi,a)

connector Pull-Dataflow

role ProducerCopy$_1$ = $\overline{dataReady} \rightarrow$ Producer
[] request?x $\rightarrow \overline{send!x} \rightarrow$ Producer $\sqcap$ §

role ProducerCopy$_2$ = $\overline{dataReady} \rightarrow$ Producer
[] request?x $\rightarrow \overline{send!x} \rightarrow$ Producer $\sqcap$ §

role ConsumerCopy$_1$ = dataReady $\rightarrow \overline{request!x} \rightarrow$ receive?x $\rightarrow$ Consumer [] §

role ConsumerCopy$_2$ = dataReady $\rightarrow \overline{request!x} \rightarrow$ receive?x $\rightarrow$ Consumer [] §

$$\textbf{glue} = \begin{cases} ProducerCopy_1.dataReady[]ProducerCopy_2.dataReady \\ \rightarrow (\overline{ConsumerCopy_1.dataReady} \parallel \overline{ConsumerCopy_2.dataReady}) \\ \rightarrow (ConsumerCopy_1.request!x \; [] \; ConsumerCopy_2.request!x) \\ \rightarrow (\overline{ProducerCopy_1.request?x} \parallel \overline{ProducerCopy_2.request?x}) \\ \rightarrow (ProducerCopy_1.send!x \; [] \; ProducerCopy_2.send!x) \\ \rightarrow (\overline{ConsumerCopy_1.reveive?x} \parallel \overline{ConsumerCopy_2.reveive?x}) \\ \rightarrow \textbf{glue} \; [] \; \S \end{cases}$$

connector copyConnect

role Copy$_{1,2}$

glue = Copy$_1$

$$\text{Copy}_1 = \overline{Copy_1.isAlive} \rightarrow \begin{cases} Copy_1 & \textbf{when } Copy_1.ImAlive \text{ within t s} \\ Copy_1Failure & \textbf{when } \neg Copy_1.ImAlive \text{ within t s} \end{cases}$$

$$\text{Copy}_2 = \overline{Copy_2.isAlive} \rightarrow \begin{cases} Copy_2 & \textbf{when } Copy_2.ImAlive \text{ within t s} \\ Copy_2Failure & \textbf{when } \neg Copy_2.ImAlive \text{ within t s} \end{cases}$$

$$\text{Copy}_1\text{Failure} = \begin{cases} \overline{Copy_1.reset} \rightarrow \overline{Copy_2.wakeUp} \\ \rightarrow \overline{Copy_1.sleep} \rightarrow Copy_2 \end{cases}$$

$$\text{Copy}_2\text{Failure} = \begin{cases} \overline{Copy_2.reset} \rightarrow \overline{Copy_1.wakeUp} \\ \rightarrow \overline{Copy_2.sleep} \rightarrow Copy_1 \end{cases}$$

instances Precon1, Precon2: PRECON
Alarm1, Alarm2: ALARM
Precon2Alarm: PullDataflow
Alam2Alarm, Precon2Precon: copyConnect

attachements Precon1.ALARMOutput **as** Precon2Alarm.ProducerCopy$_1$
Precon2.ALARMOutput **as** Precon2Alarm.ProducerCopy$_2$
Precon1.CopyLink **as** Precon2Precon.Copy$_1$
Precon2.CopyLink **as** Precon2Precon.Copy$_2$
Alarm1.PRECONInput **as** Precon2Alarm.ConsumerCopy$_1$
Alarm2.PRECONInput **as** Precon2Alarm.ConsumerCopy$_2$
Alarm1.CopyLink **as** Alarm2Alam.Copy$_1$
Alarm2.CopyLink **as** Alarm2Alam.Copy$_2$

C.2 The Observer Pattern

C.2.1 Initial Wright Specification

component Acquisition Unit

port ToIMS = $\overline{transmitted!si} \rightarrow$ **ToIMS**

port FromIMS = receive?si $\rightarrow$ **FromIMS**

computation = (AcquireAnalog [] AcquireDigital) [] Update **where**

$$\text{AcquireAnalog} = \begin{cases} Update(si.value) & \\ \rightarrow \overline{ToIMS.transmitted!si} & \\ \rightarrow \textbf{computation} & \textbf{when} \quad s.value \neq si.value \\ & \wedge si.Type = Analog \\ & \wedge si.Status = on \end{cases}$$

$$\text{AcquireDigital} = \begin{cases} Update(si.value) & \\ \rightarrow \overline{ToIMS.transmitted!si} & \\ \rightarrow \textbf{computation} & \textbf{when} \quad s.value \neq si.value \\ & \wedge si.Type = Digital \\ & \wedge si.Status = on \end{cases}$$

Update = FromIMS.receive?si $\rightarrow$ Synchronize(si) $\rightarrow$ **computation**

component IMS

port ToAcquisition = $\overline{transmitted!si} \rightarrow$ **ToAcquisition**

port FromAcquisition = receive?si $\rightarrow$ **FromAcquisition**

$$\textbf{computation} = \begin{cases} SwitchSensorOff \ [] \ SwitchSensorOn \\ []ValidateData \ [] \ UnValidateData[]Update \end{cases} \quad \textbf{where}$$

$$\text{SwitchSensorOff} = \begin{cases} TurnOff(s) \rightarrow Update(si.status) & \\ \rightarrow \overline{ToAcquisition.transmitted!si} & \textbf{when} \quad \neg s.WorkingProperly \\ & \wedge s.Status = on \end{cases}$$

$$\text{SwitchSensorOn} = \begin{cases} TurnOn(s) \rightarrow Update(si.status) & \\ \rightarrow \overline{ToAcquisition.transmitted!si} & \textbf{when} \quad s.WorkingProperly \\ & \wedge s.Status = off \end{cases}$$

$$\text{ValidateData} = \begin{cases} Update(si.consistent) & \\ \rightarrow \overline{ToAcquisition.transmitted!si} & \textbf{when} \quad s.value \notin PossibleRange \\ & \wedge \neg si.consistent \end{cases}$$

$$\text{UnValidateData} = \begin{cases} Update(si.consistent) & \\ \rightarrow \overline{ToAcquisition.transmitted!si} & \textbf{when} \quad s.value \in PossibleRange \\ & \wedge si.consistent \end{cases}$$

Update = FromAcquisition.receive?si $\rightarrow$ Synchronize(si) $\rightarrow$ **computation**

connector Push-Dataflow

role Producer = $\overline{provide!x}$ → Producer ⊓ §

role Consumer = retrieve?x → Consumer [] §

glue = Producer.provide!x → $\overline{Consumer.retrieve?x}$ → **glue** [] §

instances Acq: Acquisition Unit
Ims: IMS
Ims2Acq: PushDataflow
Acq2Ims: PushDataflow

attachements Acq.ToIMS **as** Acq2Ims.Producer
Ims.FromAcquisition **as** Acq2Ims.Consumer
Acq.FromIMS **as** Ims2Acq.Consumer
Ims.ToAcquisisition **as** Ims2Acq.Producer

C.2.2 Resulting Wright Specification

component Acquisition Unit

$$\textbf{port}\ \text{SubjectLink} = \begin{cases} \overline{SetState!x} \rightarrow \textbf{SubjectLink} \\ [\,]\ Notify \rightarrow \overline{GetState!si} \rightarrow SendState?si \rightarrow \textbf{SubjectLink} \end{cases}$$

computation = compute [] Update **where**
compute= AcquireAnalog [] AcquireDigital

$$\text{AcquireAnalog} = \begin{cases} Update(si.value) & \\ \rightarrow \overline{SubjectLink.SetState!si} & \\ \rightarrow \textbf{computation} & \textbf{when} \quad s.value \neq si.value \\ & \qquad \wedge si.Type = Analog \\ & \qquad \wedge si.Status = on \end{cases}$$

$$\text{AcquireDigital} = \begin{cases} Update(si.value) & \\ \rightarrow \overline{SubjectLink.SetState!si} & \\ \rightarrow \textbf{computation} & \textbf{when} \quad s.value \neq si.value \\ & \qquad \wedge si.Type = Digital \\ & \qquad \wedge si.Status = on \end{cases}$$

$$\text{Update} = \begin{cases} Subject.Notify \rightarrow \overline{Subject.GetState!x} \\ \rightarrow Subject.SendState?y \rightarrow SynchronizeValue \\ \rightarrow \textbf{computation} \end{cases}$$

component IMS

$$\textbf{port}\ \text{SubjectLink} = \begin{cases} \overline{SetState!x} \rightarrow \textbf{SubjectLink} \\ [\,]\ Notify \rightarrow \overline{GetState!si} \rightarrow SendState?si \rightarrow \textbf{SubjectLink} \end{cases}$$

computation = compute [] Update **where**

$$\text{compute} = \begin{cases} SwitchSensorOff\ [\,]\ SwitchSensorOn \\ [\,]ValidateData\ [\,]\ UnValidateData \end{cases}$$

$$\text{SwitchSensorOff} = \begin{cases} TurnOff(s) \rightarrow Update(si.status) \\ \rightarrow \overline{SubjectLink.SetState!si} \qquad \textbf{when} \quad \neg s.WorkingProperly \\ \qquad \wedge s.Status = on \end{cases}$$

$$\text{SwitchSensorOn} = \begin{cases} TurnOn(s) \rightarrow Update(si.status) \\ \rightarrow \overline{SubjectLink.SetState!si} \qquad \textbf{when} \quad s.WorkingProperly \\ \qquad \wedge s.Status = off \end{cases}$$

$$\text{ValidateData} = \begin{cases} Update(si.consistent) \\ \rightarrow \overline{SubjectLink.SetState!si} \quad \textbf{when} \quad s.value \notin PossibleRange \\ \qquad \wedge \neg si.consistent \end{cases}$$

$$\text{UnValidateData} = \begin{cases} Update(si.consistent) \\ \rightarrow \overline{SubjectLink.SetState!si} \quad \textbf{when} \quad s.value \in PossibleRange \\ \qquad \wedge si.consistent \end{cases}$$

$$\text{Update} = \begin{cases} Subject.Notify \rightarrow \overline{Subject.GetState!x} \\ \rightarrow Subject.SendState?y \rightarrow SynchronizeValue \\ \rightarrow \textbf{computation} \end{cases}$$

component Subject

$$\textbf{port}\ \text{Observer}_{1,2}\text{Link} = \begin{cases} SetState?x \rightarrow \overline{Notify} \rightarrow \textbf{Observer}_{\mathbf{1,2}}\textbf{Link} \\ [\,]\ GetState?x \rightarrow \overline{SendState!y} \rightarrow \textbf{Observer}_{\mathbf{1,2}}\textbf{Link} \end{cases}$$

computation = SetState → Notify [] GetState **where**

SetState=(Observer$_1$.SetState?x [] Observer$_2$.SetState?x)

$$\text{Notify} = (\overline{Observer_1.Notify} \parallel \overline{Observer_2.Notify}) \rightarrow \textbf{computation}$$

$$\text{GetState} = \begin{cases} (Observer_1.GetState?x \rightarrow \overline{Observer_1.SendState!y} \\ \rightarrow \textbf{computation}) \\ [\,](Observer_2.GetState?x \rightarrow \overline{Observer_2.SendState!y} \\ \rightarrow \textbf{computation} \end{cases}$$

connector Observer-Subject Link

$$\textbf{role}\ \text{Observer} = \begin{cases} \overline{SetState!x} \rightarrow \textbf{Observer} \\ [\,]\ Notify \rightarrow \overline{GetState!x} \rightarrow SendState?y \rightarrow \textbf{Observer} \end{cases}$$

$$\textbf{role}\ \text{Subject} = \begin{cases} SetState?x \rightarrow \overline{Notify} \rightarrow \textbf{Subject} \\ [\,]\ GetState?x \rightarrow \overline{SendState!y} \rightarrow \textbf{Subject} \end{cases}$$

$$\textbf{glue} = \begin{cases} Observer.SetState!x \\ \to \overline{Subject.SetState?x} \to Notify \quad \textbf{where} \\ []Notify \end{cases}$$

$$\text{Notify} = \begin{cases} Subject.Notify \to \overline{Observer.Notify} \\ \to Observer.GetState!x \to \overline{Subject.GetState?x} \\ \to Subject.SendState!y \to \overline{Observer.SendState?y} \to \textbf{glue} \end{cases}$$

instances Acq: Acquisition Unit
Ims: IMS
SensorInformation : Subject
Ims2SensorInformation: Observer-Subject Link
Acq2SensorInformation: Observer-Subject Link

attachements Acq.SubjectLink **as** Acq2SensorInformation.Observer
SensorInformation.Observer$_1$Link **as** Acq2SensorInformation.Subject
Ims.SubjectLink **as** Ims2SensorInformation.Observer
SensorInformation.Observer$_2$Link **as** Ims2SensorInformation.Subject