Deriving Architectural Descriptions from Goal-Oriented Requirements Models

Damien Vanderveken
Axel van Lamsweerde
Dept. Ingénierie Informatique
Université catholique de Louvain
avl@info.ucl.ac.be

Dewayne Perry
Empirical Software Engineering Lab
Electrical & Computer Engineering
University of Texas at Austin
perry@ece.utexas.edu

Christophe Ponsard
CETIC Research Center
Charleroi (Belgium)
cp@cetic.be

ABSTRACT

The problem of building an architecture that satisfies the software requirements is obviously central to software engineering. By and large such building is an ad hoc, largely informal, and unsystematic process to date. One strand of research to address this challenge is based on architecture derivation from goal-oriented requirements models.

This paper builds on previous efforts in this direction and proposes a more formal approach to architectural derivation. The structural and behavioral parts of a formal dataflow architecture are derived first by use of transformation rules applied to the requirements model. The dataflow architecture is then refined by use of formal refinement patterns applied to components and connectors. Each refinement pattern is associated with a specific class of non-functional goal whose instances are found in the requirements model. The source and target languages are the KAOS requirements language and Wright architecture description language, respectively. A power plant supervision system is used as a case study to illustrate the main steps of the derivation.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Languages, Patterns; D.2.1 [**Requirements/Specifications**]: Methodologies.

Keywords

Architecture derivation, goal orientation, requirements engineering, refinement patterns, non-functional requirements, architecture description languages.

1. INTRODUCTION

Requirements engineering is concerned with the elicitation of the goals the system-to-be should satisfy, the operationalization of such goals into specifications of services and constraints, and the assignment of responsibilities for the resulting requirements to system agents such as humans, devices, and software. Architectural design is concerned with the overall organization of the software part of this system into components and interactions between them.

Requirements engineering and architectural design are highly intertwined. On the one hand, the architecture has to meet the requirements on the product's functionality and the non-functional requirements on the product's quality. On the other hand, the requirements are produced by an elaboration process faced with many alternative options that result in alternative designs. Moreover, requirements elaboration needs to take into account environment constraints that may induce specific architectural styles –such as, e.g., the physical distribution of agents, interoperability with legacy software, or installation constraints.

Some effort has been devoted recently towards better understanding of the interplay between requirements and architecture [2][6]. For example, an intermediate model can be introduced to bridge the gap between requirements-related concepts and architecture-related ones [16]. The architecture can be viewed as an emergent property of requirements integration; functional requirements could then be mapped to some high-level design using genetic transformation techniques [12]. Requirements-architecture links can also be studied in the problem frame setting; architectural constraints are then elements of the problem domain that may drive problem decomposition and recomposition [17][35].

Once the relationship between requirements and architecture is better understood, a key research challenge is to find systematic techniques for constructing a software architecture that meets the elaborated requirements. Bosch and Molin suggest an informal, iterative process for architecture elaboration based on successive evaluations and transformations of architectural drafts to meet non-functional concerns [3]. On a more formal side, correctness-preserving transformations have been proposed to refine abstract architectures into concrete ones [30].

Goal orientation appears to be a promising paradigm for defining two-way links between requirements and architectures and for exploring derivation processes. It offers a unified framework in which both functional and non-functional concerns can be integrated; refinement/abstractions links are defined precisely and provide the basis for various forms of qualitative, quantitative, or formal reasoning [23]. For example, Gross and Yu show how non-functional goals can be used to document design patterns; qualitative reasoning schemes can then be applied to select goal-matching patterns during the architectural design process [15]. In the Preskriptor process, a component refinement tree is built from requirements so that each goal is ensured by a component [4][5]. In the context of the KAOS project, we have also developed a 3-step derivation process whereby (a) components and dataflow connectors are derived first from functional requirements in a goal-oriented model, (b) the abstract dataflow architecture is refined to accommodate some architectural style that meets architectural constraints, and (c) the styled architecture is refined so as to meet the non-functional goals from the goal model [24]. The third step is based on architectural refinement patterns associated with specific non-functional goal categories.

To gain a deeper insight into the goal-directed strands of architecture derivation, Jani et al. deployed the Prescriptor and KAOS approaches on a complex system [19][20], the maintenance system of an Italian power plant [8]. The evaluation part of this work revealed some limitations of both approaches, notably, the lack of precision in the specification of the derived artefacts and the lack of coverage of architectural behavior.

This paper presents an approach to address those limitations. Our goal-oriented approach to architecture derivation is refined and extended as follows:

- the derived architecture is specified in an architecture description language (ADL),
- the derivation covers architectural behaviors,
- the refinement steps and patterns are expressed as transformation rules towards the target ADL.

ADLs provide support for explicitly modeling software components, connectors, their configurations, and constraints on the components, connectors and configurations. Beyond the level of precision gained, a formal description of the architecture is amenable to architecture-level analysis to examine whether properties of interest are satisfied [11][18].

Note, however, that an architectural model needs to be “coded” in the ADL for such analysis to be possible. Such model building and coding is far from being trivial, and is error-prone. Our approach is also aimed at supporting architects in this process.

Many ADLs have been proposed, e.g., [1][27][29]. Our choice of *Wright* among several other candidates was motivated by the fact that the same language can be used to capture architectural structures, behaviors, and styles.

The paper is structured as followed. Section 2 provides some background on the source and target of our derivation process, namely, goal-oriented requirements models in KAOS and architectural descriptions in Wright. The ENEL power plant supervision system is also introduced there as our running case study. Section 3 overviews our derivation process. Section 4 shows how a Wright dataflow architecture is derived. Rules for mapping functional requirements of KAOS operations onto Wright behavioral descriptions are discussed in Section 5. Section 6 then presents our pattern-based refinement process by focussing on two important categories of non-functional goals, namely, accuracy goals and fault tolerance goals.

2. BACKGROUND

This section briefly reviews what the input to and output from our architecture derivation process look like.

2.1 The source: goal-oriented requirements models in KAOS

We start from a multi-facet requirements model that integrates a goal model, an object model, an agent model, and an operation model. A systematic method is available for building such models, see [23].

2.1.1 The Goal Model

A *goal* is a prescriptive statement of intent about the system-to-be whose satisfaction requires the cooperation of agents from that system. *Agents* are active components playing some *role* towards goal satisfaction. They can be humans, devices, legacy software, or software-to-be components. Some agents form the software whereas others form the software environment. Functional goals refer to services to be provided whereas non-functional goals refer to quality of service. Unlike goals, *domain properties* are descriptive statements about the environment. They may refer to physical laws, organizational policies, and the like.

Goals are organized in AND/OR *refinement-abstraction hierarchies*. Higher-level goals are in general strategic, coarse-grained and involve multiple agents; lower-level goals are in general technical, fine-grained and involve fewer agents [9][10]. In such structures, *AND-refinement* links relate a goal to a set of subgoals (called *refinement*) possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. *OR-refinement* links relate a goal to a set of alternative refinement options.

Goal refinement ends when every subgoal is *realizable* by some individual agent assigned to it; the goal must be formulated in terms of conditions that are monitorable or controllable by the agent [25]. A *requirement* is a terminal goal under responsibility of an agent in the software-to-be; an *expectation* is a terminal goal under responsibility of an agent in the environment.

Goals prescribe intended behaviors; they are optionally formalized in a real-time linear temporal logic. Keywords such as *Achieve*, *Avoid*, *Maintain* are used to name goals according to the temporal behavior pattern they prescribe.

Figure 1 shows a goal model fragment for our running case study, the ENEL power plant supervision system [8]. The leaf goal *AlarmRaisedWhenFaultDetected* may be annotated with the following temporal logic assertion stating that a single alarm should be issued within *T* time units when a fault is detected:

$$\forall \text{ f: Fault : f.detected } \Rightarrow \Diamond_{\leq T} (\exists! \text{ al: Alarm: Reporting (al, f)}$$

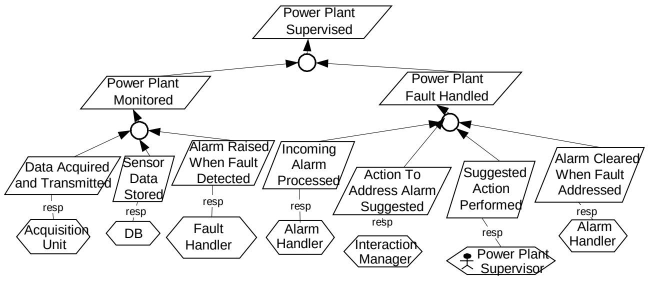
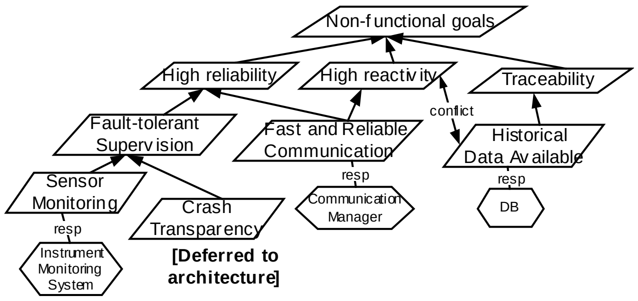


Figure 1 - ENEL supervision: functional goals

Non-functional goals are additional constraints on the way the system should achieve its functional goals. They guide the goal refinement process and, in particular, the selection of “best” options among possible alternatives. In Figure 2, the goal *HistoricalDataAvailable* is seen to be assigned to a database agent which might call for a repository-based design for detecting faults from the data collected (rather than an event-based design). This goal may however conflict with real-time requirements [22]. Some goals may be linked to well-identified agents – such as the Instrument Monitoring System (IMS) or the Communication Manager. Other assignments are deferred to the architectural design phase. As seen later in the paper, non-functional goals will drive the architectural refinement process.



Non-functional goals
High reliability
High reactivity
Traceability
conflict
Fault-tolerant Supervision
Fast and Reliable Communication
Historical Data Available
resp
Sensor Monitoring
Crash Transparency
Communication Manager
DB
Instrument Monitoring System
[Deferred to architecture]

Figure 2 - ENEL supervision: non-functional goals

2.1.2 The Object Model

Objects are incrementally derived from goal specifications to produce a structural model of the system (represented by UML class diagrams), see Figure 3. Objects have *states* defined by the values of their attributes and associations to other objects. They are passive (entities, associations, events) or active (agents). Agents are related together via their interface made of the object attributes and associations they *monitor* and *control*, respectively [25]. In the above formalization of the goal *AlarmRaisedWhenFaultDetected*, the attribute *detected* is attached to the *Fault* entity. *Reporting* is an association between the *Alarm* and *Fault* entities.

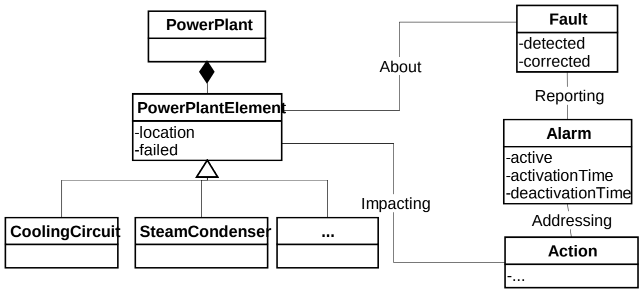


Figure 3 - ENEL supervision: object model

2.1.3 The Agent Model

The agent model defines the various agents forming the system together with their responsibility links to the goal model and monitoring/control links to the obect model. For example, the above goal *AlarmRaisedWhenFaultDetected* turns to be a requirement assigned to the *AlarmHandler* agent. The latter must be able to monitor the *detected* attribute on *Fault* and control the *Reporting* relationship.

A useful artefact generated from the agent model is the agent interface view, see Fig. 4. This view shows the flow of monitored/controlled information among the agents. Our architecture derivation process will rely on this view to build a first architectural sketch.

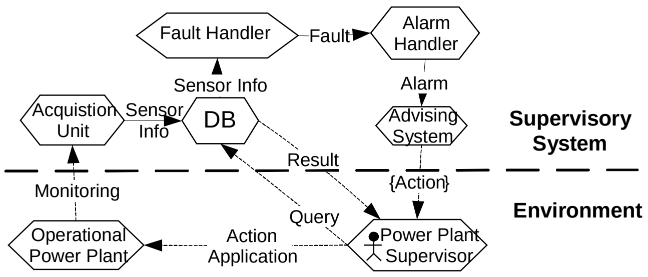


Figure 4 - Interface view of the agent model

2.1.4 The Operation Model

Goals are *operationalized* into specifications of operations to achieve them [9][26]. An *operation* is an input-output relation over objects. Operation applications define state transitions along the behaviors prescribed by the goal model.

When specifying an operation, a distinction is made between domain pre/postconditions and additional pre-, post- and trigger conditions required for achieving some underlying goal. A pair *(domain precondition, domain postcondition)* captures the elementary state transitions defined by operation applications in the domain. A *required precondition* for some goal captures a permission to perform the operation when the condition is true. A *required trigger conditio*n for some goal captures an obligation to perform the operation when the condition becomes true provided the domain precondition is true. A *required postcondition* defines some additional condition that any application of the operation must establish to achieve the corresponding goal.

For example, the following operations operationalize the goals *AlarmRaisedWhenFaultDetected* and *AlarmClearedWhenFault-Handled* in Figure 1, respectively.

Operation RaiseAlarm
  Input: f:Fault; **Output:** al:Alarm, Reporting
  DomPre ¬ Reporting(al,f)
  DomPost Reporting(al,f) ∧ al.active
  ReqTrig for *AlarmRaisedWhenFaultDetected:* @f.detected

Operation ClearAlarm
  Input: f:Fault**,** Reporting; **Output:** al:Alarm
  DomPre: al.active
  DomPost: ¬ al.active
  ReqTrig for *AlarmCleardWhenFaultHandled:*
    @f.corrected ∧ Reporting(al,f)

2.2 The target: architecture descriptions in Wright

The Wright language allows architects to formalize architectural structures, behaviors, and styles. An architecture is described in terms of components, connectors, and configurations [1].

A *component* is characterized by an interface and a computation. An *interface* consists of a number of ports. Each *port* captures an interaction in which the component may participate; it describes a behavior at that particular point of interaction. A *computation* describes the component behavior. It carries out the interactions described by the ports and specifies how they compose together.

A *connector* is characterized by a set of roles and a 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. Similarly to the computation of a component, the glue of a connector carries out the interactions described by the roles and specifies how they compose together

To form an architecture, the components and connectors must be combined into a configuration. A *configuration* is a collection of component instances combined through connectors. It defines instances and attachments. *Instances* are needed as multiple components or connectors can be of the same type. *Attachments* describe the software topology by linking components instances via connector instances. Hierarchical descriptions are supported. The overall behavior of an architectural configuration is defined by the composition of the individual behaviors of its components, according to the ordering and data transfer prescribed by the glue of the connectors to which they are attached.

The formal notation used to describe behaviors 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 have associated data (as in *e?x* and *e!x*, representing data input and output, respectively). The simplest process, STOP, is the one that engages in no events. The symbol *§* is used to denote the successful termination of a process. 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 denotes 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 denotes the parallel execution of processes *P* and *Q*.

Conditions. Different process behaviors can be expressed by use of the **when** operator:

$$P_v = \begin{cases} Q & \textbf{when } A(v) \\ R & \textbf{otherwise} \end{cases}$$

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 used in a process expression through the **where** operator.

Tools are available for analyzing Wright architectures. Classic checks include the detection of deadlocks, starvation, and race conditions. Another important facility is that of checking whether attached ports and roles are behaviorally compatible. As Wright uses a subset of CSP, the FDR model checker can be used [13].

3. OVERVIEW OF THE ARCHITECTURE DERIVATION PROCESS

Our approach integrates previously separate efforts to derive architectural structures in a goal-oriented way [24][20], and extends them to the derivation of formal ADL descriptions. The main steps are outlined here before being detailed in the next sections.

1. *Deriving the structural part of an abstract architecture:* The interface view of the KAOS agent model is mapped on dataflow components/connectors. The process is based on monitoring/control links among agents from the requirements model.
2. *Deriving architecture behaviors:* The operationalization view from the requirements model is used to obtain the behavioral part of the architecture description. The process is based on a set of heuristic derivation rules.
3. *Refining the architecture:* Non-functional goals are incrementally taken into account to refine architectural components/connectors. The process is based on refinement patterns associated with specific non-functional goal categories.

The architecture is described semi-formally with box-and-arrow diagrams but also formally in Wright, starting from the first step and through all refinements. The derivation rules are made precise on those formal notations.

It is important to note that some global design decisions may already have been made at the requirements engineering stage [24]. This is due to the space of alternative options in which the requirements engineer has to make choices.

- A goal is often refinable into alternative AND-combinations of subgoals [10].
- A risk is often reducible through alternative countermeasures [23].
- A conflict is often manageable through alternative resolutions [22].
- A “terminal” goal in the goal refinement process is often assignable to alternative agents [25].
- There may be alternative choices on the *granularity* of such agents – from coarser-grained agents, assigned to coarser-grained goals, to finer-grained agents assigned to finer-grained subgoals of the coarser-grained goals. In fact, it is often convenient to structure agents in aggregation hierarchies that reflect the structure of the environment [9], as experienced in [5].

For each type of alternative option, decisions need to be made at requirements engineering time based on higher contribution to non-functional requirements, reduction of risks, and resolution of conflicts [23]. Such decisions often have a global impact on the architecture as different choices result in different designs [32]. In this paper we are at the stage where such decisions have been made. In particular, a preliminary granularity has been decided for the agents to be mapped on architectural components.

4. DERIVING AN ABSTRACT DATAFLOW ARCHITECTURE

Our starting point is the KAOS agent interface view that describes the data dependency links among agents.

As introduced before, an agent depends on another for some attribute/association from the object model if it monitors that attribute/association and the other controls it. Each such dependency defines a *data flow*.

The following rules describe our mapping on a structural Wright description.

Rule 1. *Each software agent in the KAOS agent model with n data flows from monitor/control links is mapped to a Wright component with n ports:*

```
component CW
    port p1
    ...
    port pn
```

Rule 2. *Each data flow from monitor/control links in the KAOS agent model is mapped to a Wright connector type* Dataflow *with two roles:*

```
connector Dataflow
    role Producer
    role Consumer
```

Rule 3. *Let AG be an agent in the KAOS agent model and let CW be its corresponding Wright component. Each instance variable ag of type AG in KAOS assertions is mapped to a corresponding instance of type CW in Wright specifications.*

Rule 4. *For each data flow instance in the KAOS model an instance of the* Dataflow *connector type is defined in Wright specifications.*

Rule 5. *Let AG_1, AG_2 be two agents in the KAOS model linked by a data flow. Let CW_1, CW_2 be the corresponding Wright components. Let Output and Input be the ports used by each one to interact with the other. Let DFW be the corresponding Wright dataflow connector with its two Producer and Consumer roles. The corresponding instances cw_1, cw_2 and dfw are attached as follows:*

```
attachments    cw1.Output as dfw.Producer
               cw2.Input as dfw.Consumer
```

The above rules yield the following architectural configuration from the agents *FaultHandler* and *AlarmHandler* in Fig.4:

```
configuration FaultHandler-AlarmHandler
    component FaultHandler
        port AlarmHandlerOutput
    component AlarmHandler
        port FaultHandlerInput
    connector Dataflow
        role Producer
        role Consumer
    instances    fh: FaultHandler
                 al: AlarmHandler
                 FaultHandlerToAlarmHandler: Dataflow
    attachments
        fh.AlarmHandlerOutput as
            FaultHandlerToAlarmHandler.Producer
        al.FaultHandlerInput as
            FaultHandlerToAlarmHandler.Consumer
```

5. DERIVING BEHAVIORS

We now address the derivation of the behavioral part of Wright components/connectors from the behaviors prescribed in the KAOS operation model. A set of heuristic derivation rules is presented for guiding the elaboration of the Wright specification. These heuristics emerged from multiple examples in the ENEL case study. Each heuristic will be introduced intuitively, described precisely, and illustrated.

We first consider the behavior of connectors. The behavior of a component will then be derived from the specification of the operations assigned to it.

5.1 Connectors

Dataflow connectors capture the interaction between a *Producer* component providing some data and a *Consumer* component needing these data. Two alternative behaviors can be considered. In the *push* behavior, the producer initiates an event with data attached to it each time new data are produced. In the *pull* behavior, the producer notifies the consumer each time new data are produced; the consumer has 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 = provide!x → Producer ⊓ §
    role Consumer = retrieve?x → Consumer [] §
    glue = Producer.provide!x → Consumer.retrieve?x
              → glue [] §
connector Pull-Dataflow
    role Producer = dataReady → Producer
                    [] request?x → send!x → Producer ⊓ §
    role Consumer = dataReady → request!x → receive?x
                    → Consumer [] §
    glue = Producer.dataReady → Consumer.dataReady
           → Consumer.request!x → Producer.request?x
           → Producer.send!x → Consumer.receive?x → glue [] §
```

The choice between the "push" and "pull" alternatives is a design decision that may be linked non-functional goals such as the following:

- *Reactiveness:* a push connector is more appropriate for meeting hard real-time requirements.
- *Load*: if the consumer is subject to overload, it would be better to "pull" when it is ready. If the producer cannot keep the produced data for lack of resources, a "push" approach may be more appropriate.

5.2 Components

The Wright specification of the behavior of a component is derived from the specification of the operations the corresponding agent has to perform in the KAOS operation model. We first discuss how such operations are individually mapped to the component. The inference of the control flow among operations within the component is briefly discussed next.

5.2.1 Mapping individual KAOS operations

Pre/post and trigger conditions

Two constructs can be used in Wright to represent pre-, post-, and trigger conditions: events and logical conditions preceded by the **when** operator. This operator prescribes different behaviors dependent on the truth value of some condition; it can therefore be used to translate pre- and trigger conditions only. Postconditions must therefore be translated into Wright events. If a precondition also appears as postcondition of another operation within the same component, the precondition will be translated into an event as well. If this is not the case, two possibilities remain.

- A precondition is a postcondition of an operation performed by another component. This requires some transmission from that other component to notify satisfaction of the condition. If the transmission is an event coming from one of the ports, the precondition will be represented as an event.
- A precondition is only a precondition of that particular operation. In this case the **when** operator will be used. The precondition then typically characterizes the state of an internal variable or some property of the environment.

The above discussion leads to the following heuristic rules.

Rule 6. *Let C be a component with operations $Op_1,\ldots,Op_n$. The postconditions of $Op_1,\ldots,Op_n$ are expressed in Wright by events.*

Rule 7. *Let C be a component with operations $Op_1,\ldots,Op_n$. The pre- and trigger conditions of $Op_1,\ldots,Op_n$ can be expressed either by events or by conditions 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 to some property of the environment. Events are used otherwise.*

Let us illustrate this on the *AlarmHandler* in our ENEL case study. This component is responsible for the operations *RaiseAlarm* and *ClearAlarm* described in Section 2.1. The precondition of *ClearAlarm* will be expressed as an event *Corrected?f* and its postcondition as *Inactive!al*. As the precondition refers to an internal constraint binding the output alarm *al* to the input fault *f*, it will be expressed as **when** condition *Reporting(al,f)*.

Inputs and Outputs

Wright has CSP-like constructs for data transmission that can be used to express incoming and outgoing parameters. Data can be associated with events though *?d* for reception and *!d* for transmission. The dataflow connector presented in Section 5.1 will ensure that the value is "pushed" or "pulled" between the components. A simple heuristic rule is therefore the following.

Rule 8. *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. An event e_1 of form $p_1.e_1!d$ must be found in the specification of C_1 and an event e_2 of form $p_2..e_2?d$ must be found in the specification of C_2.*

For example, in *AlarmHandler*, events *Detected?f* and *Corrected?f* should appear for the incoming fault from the *FaultHandler* fault detector.

Forming a Wright process for a KAOS operation

The Wright process is composed by linking the operation's trigger-, pre-, and post-conditions translated according to the previous heuristics. As pre- and trigger conditions "precede" post-conditions, there should be a sequential order between them. When expressed as events they are therefore linked to the corresponding postconditions using the Wright → construct. The pre- and trigger conditions expressed by a condition are added through the **when** operator. Three distinct cases can be identified and are detailed in the following heuristic rule.

Rule 9. *Let Op be an operation. Let pre be the event expressing pre- and trigger conditions (if introduced). Let cond be the condition expressing preconditions (if introduced). Let post be the event expressing postconditions. The Wright specification of the process representing Op is as follows:*

1. if pre is expressed by events: pre → post,

2. if pre is expressed by conditions: post **when** *cond,*

3. if pre mixes events and conditions: pre → post **when** *cond.*

By convention, the same names will be used for the KAOS operation and the Wright process.

The above heuristic rules yield the following Wright specification for the *AlarmHandler* component:

component AlarmHandler
 port FaultHandler-in = Detected?f []Corrected?f → FaultHandler-in
 computation: (RaiseAlarm [] ClearAlarm) → **computation**
  where
   RaiseAlarm= FaultHandler-in.Detected?f → $\overline{\text{Reporting!(al,f)}}$
   ClearAlarm = FaultHandler-in.Corrected?f → $\overline{\text{Inactive!a}}$
         when Reporting(al,f)

5.2.2 Infering the intra-component control flow among operations

In the above example, the computation was expressed as a deterministic choice between all component operations. This relies on the assumption that at most one operation will be enabled at a given time. This happens to be true in the example as a fault cannot be detected and corrected at the same time.

In general, some additional control flow is needed to compose operations within a computation. Inferring it requires more complex rules as the KAOS specification does not constrain the ordering of operations explicitly; it does so implicitly through pre, trigger-, and postconditions.

The general idea is to let one operation *Op1* precede another operation *Op2* when *Op2* is *logically dependent* on *Op1*, for example, when the postcondition of *Op1* logically implies the precondition of *Op2*, with direct sequencing through the Wright → operator when the postcondition of *Op1* logically implies the trigger condition of *Op2*the trigger condition of the other has some corresponding form. (The principle is similar to the one used in AI planning.) Parallel composition of *Op1* and *Op2* is introduced through the Wright || operator when there is no such logical dependency and no overlap between the lists of output variables declared in their respective KAOS *Ouput* clause – in other words, the operations do not interfere. For operations that are not composed sequentially or in parallel, *Op1* is composed with *Op2* through the Wright [] operator when at least one of their preconditions refers to the environment; otherwise the $\sqcap$ operator is used.

Lack of space prevents us from detailing the derivation rules making the above general idea more precise. Details can be found in [37].

6. PATTERN-BASED REFINEMENT OF THE ARCHITECTURE

It has long been recognized that architectural design has a major impact on non-functional goals [33]. Many such goals impose constraints on interactions between components. For example,

- security goals restrict interactions so as to limit information flows along communication channels,

- accuracy goals require interactions to maintain a consistent state between related objects,
- usability requirements put static and dynamic constraints on interactions with environment components.

The key principle underpinning architectural refinement is to "inject" non-functional goals from the goal model within component interactions (connector refinement) or within single components (component refinement). The general procedure is made more precise as follows [24].

- For each non-functional leaf goal *NFG* in the goal model:
  - identify all connectors and components that are specifically concerned by *NFG*;
  - instantiate *NFG's* spec to those connectors and components.
- For each such connector or component:
  refine it to meet the instantiated *NFG* spec by instantiation of an architectural refinement pattern associated with *NFG*'s goal category.

A number of such patterns were suggested in [24] for categories of nun-functional goals such as accuracy goals, availability goals, fault tolerance goals, interoperability goals, information hiding goals, cohesion goals, etc. Those patterns were just sketched there quite informally.

Here we show how they can be made much more precise through formalization in Wright, and illustrate their use in the ENEL case study. For obvious space reasons we limit ourselves to two (non-trivial) refinement patterns. Each pattern is described by a rewrite rule taking the form:

(driving non-functional goal,
constrained components/connectors)
→ resulting architectural fragment

6.1 Maintaining accurate interaction through observer-based refinement

Fig. 5 shows a graphical rewrite rule for a refinement pattern to achieve data accuracy among interacting components.

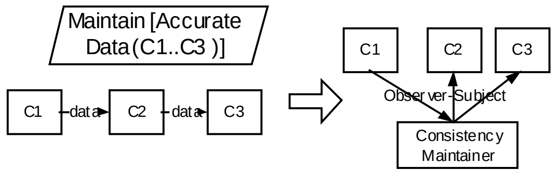


Figure 5 – Observer-based refinement for data accuracy

Driving non-functional goal: Maintain the consistency between multiple representations of some common master concept.

Constrained components/connectors: Components *C1*, *C2*, ..., *Cn* are linked through dataflow connectors; every component may own a specific representation for the data.

Resulting architectural fragment: The data flow is redirected through a new component whose role is to maintain data consistency. The components are no longer interconnected together (see Fig. 5). They play an *Observer* role whereas the Consistency Maintainer (CM) plays a *Subject* role [14].

- CM's interface provides a method to enable components to obtain its state and update their state.
- Whenever a component modifies its data it updates CM's state. Whenever it receives a notification from CM it gets the new state.

In the Wright formalization that follows, components playing the *Observer* role have the ability to trigger a change through *SetState*; CM will update itself and issue notifications to the other observing components which then update themselves accordingly. This behavior is captured in the glue specification of the connector.

component ConsistencyMaintainer
  port $\text{Obs}_{1,2}$Link =
    SetState?x → Notify → $\text{Obs}_{1,2}$Link
  [] GetState?x → $\overline{\text{SendState!y}}$ → $\text{Obs}_{1,2}$Link
  computation = SetState → $\overline{\text{Notify}}$ [] GetState **where**
    SetState = (Obs_1.SetState?x [] Obs_2.SetState?x)
    Notify = ($\overline{\text{Obs1.Notify}}$ || $\overline{\text{Obs2.Notify}}$) → **computation**
    GetState =
      (Obs_1.GetState?x → Obs1.SendState!y → **computation**)
    [] (Obs_2.GetState?x → $\overline{\text{Obs2.SendState!y}}$ → **computation**)

component Component
  port SbjLink =
    SetState!x → **SbjLink**
  [] Notify → GetState!x → SendState?y → **SbjLink**
  computation = compute [] Update **where**
    compute = … ToOtherObs.receive?x…→ Modify(x)
      → SbjLink.SetState!x → ToOtherObs.Send!x…
    Update = SbjLink.Notify → SbjLink.GetState!x
      → SbjLink.SendState?y→ SyncValue→
computation

connector Observer-Subject Link
  role Observer = $\overline{\text{SetState!x}}$ → **Observer**
    [] Notify → $\overline{\text{GetState!x}}$ → SendState?y→ **Observer**
  role Subject = SetState?x → Notify → **Subject**
      [] GetState?x → $\overline{\text{SendState!y}}$ → **Subject**
  glue = (Observer.SetState!x → Subject.SetState?x → **glue)**
      [] (Subject.Notify → Observer.Notify
        → Observer.GetState!x → Subject.GetState?x
        → Subject.SendState!y → Observer.SendState?y
        → **glue)**

The above refinement pattern was applied to the ENEL case study to manage the consistency between the *AcquisitionUnit* and *IMS* components, as shown at the bottom of Fig. 7. The formalisation can be found in [19][37].

6.2 Achieving fault-tolerance through redundancy-based refinement

Fig. 6 shows a graphical rewrite rule for a refinement pattern to achieve fault-tolerant communication among interacting components.

Driving non-functional goal: Maintain a fault-tolerant communication scheme in a transparent way.

Constrained components/connectors: Components *C1*, *C2* are linked through some connector; each of them is subject to faults.

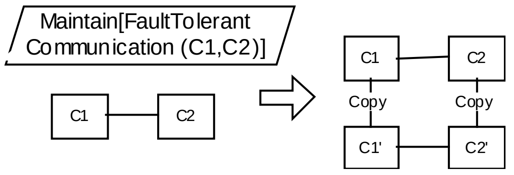


Figure 6 – Refinement pattern for fault-tolerant communication

Resulting architectural fragment: In the spirit of [7], transparent fault tolerance is achieved by first ensuring fault tolerance and then masking it. The refined architectural fragment is based on a two-phase organization.

Detection phase

1. *Introduction of a fault detection mechanism* to detect process crashes or component misbehavior.
2. *Maintenance of a pair of copies*. There is no no master/slave relationship among copies. At any given time, only one copy is active while the other is ready to take over in case of failure (transiently it may be busy recovering from its own failure). The two copies behave exactly the same way; they have the same Wright specification.

Correction phase:

1. *Introduction of a reset procedure* to restart any component that may have crashed.
2. *Switch in case of failure.* When a component goes down, its copy has to take on. The change of operating component may not result in any processing error or loss of information. The crashed component must also be reset.
3. *Transparency of switching*. The switch in operating component has to be done in a completely transparent way to other interacting components. The latter should not notice that a switch has occurred. Interacting components should also not be aware of duplicate component copies.

The core of the refined architecture is the *Copy* connector and, in particular, how and when this connector is activated. The following Wright excerpt shows the reconfiguration logic in this connector (the synchronization logic is not shown here for lack of space). The glue specification shows that when the "heart beat" is lost from the active copy, it is reset and replaced by the mirror copy which becomes the main copy while the reset one becomes the new mirror ready to take over.

connector Copy
  role $\text{Copy}_{1,2}$
  glue = Copy_1

$$\text{Copy}_i = \overline{\text{Copy}_i.\text{isAlive}} \rightarrow \begin{cases} \text{Copy}_i \text{ when } \text{Copy}_i.\text{ImAlive} & \text{(within T s)} \\ \text{Copy}_i\text{Failure when } \neg\text{Copy}_i.\text{ImAlive} & \text{(within T s)} \end{cases}$$

$$\text{Copy}_i.\text{Failure} = \text{Copy}_i.\text{reset} \rightarrow \text{Copy}_{\text{mirror(i)}}.\text{wakeUp} \rightarrow \overline{\text{Copy}_i.\text{sleep}} \rightarrow \text{Copy}_{\text{mirror(i)}}$$

The above pattern was applied to *FaultHandler* and *AlarmHandler* in the ENEL case study to achieve the final architecture shown in Fig. 7. The full formalisation and additional design details to achieve transparent fault-tolerant interaction (here between FaultHandler and AlarmHandler) can be found in [19][37]. Also note that a broker is introduced in Fig. 7 for managing communication. It was obained by use of another pattern not discussed in this paper.

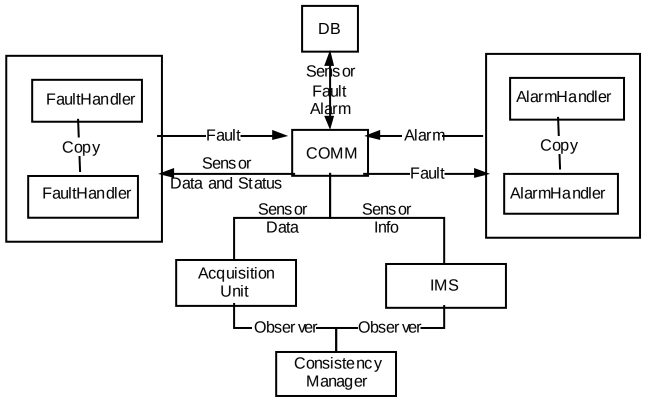


Figure 7 – Final architecture of the ENEL power plant supervision system after pattern-based refinement

7. CONCLUSION

In spite of increasing interest in bridging the gap between requirements and architecture, very little support is available for guiding architects in the elaboration of an architecture guaranteed to meet the functional and non-functional requirements elaborated by requirements engineers. Techniques are now available for eliciting, specifying, and analyzing requirements. Techniques are also available for specifying and analyzing architectures on formal grounds. But how do we get from such requirements to such architectures in a systematic way? At best architects can reuse architectural styles and patterns whose underlying requirements are in general formulated in vague terms compared with the level of precision found in the requirements specification.

Our approach addresses this challenge. The requirements model integrates multiple models that are built in a systematic, incremental way. The goal model is used to derive the object, agent, and operation models [23]. The agent model is used to derive the structural part of a dataflow architecture. The operation model is used to derive the behavioral part of that architecture. The non-functional goals from the goal model are then used to drive the refinement of components and connectors. The refined architecture is described in an ADL; it can therefore be analyzed formally against requirements from the goal model. In case of problem one may then backtrack to the architecture or to the requirements model and repeat the process. As requirements may have evolved in the meantime a new cycle may be needed anyway.

A continuum of model elaboration is thereby made possible in the perspective of requirements-architecture co-design. Alternative options are explored and assessed from the beginning. A continuum of analysis tools can be used accordingly to support the process from goal-oriented requirements analyzers[1], animators and refinement checkers [34], to CSP tools such as PROBE and FDR [13].

[1] http://www.objectiver.com

Full traceability is ensured from the high-level system goals to the architectural components and connectors. In particular, the architectural refinement of connectors/components is explicitly linked to the non-functional goals the refinement aims to achieve. As a consequence, *architectural views* can be associated with specific non-functional features, e.g., security views or fault tolerance views of he architecture. Such views might be easily extracted through query facilities such as the one provided by the Objectiver tool.

Our approach has been validated on a non-trivial system, the power plant supervisory system of an Italian electricity company for which sufficient documentation was available to carry out the derivation at a fine-grained level of detail.

Our work represents a first step towards the formal derivation of software architectures. There are many aspects that need be further worked out.

Our mapping rules have not yet been formally proved to be semantics-preserving, in the sense that the set of architectural behaviors is a subset of the set of behaviors prescribed by the goal model. Some deep examination of the formal semantics of the source and target languages would be necessary which falls outside the scope of this paper. At this stage we rely on correctness arguments. We can also run verification tools [13] on the Wright specification to check for consistency with properties from the goal model. One specific problem with Wright is its lack of support of timing constraints. Some level of expressiveness may thus be lost when translating KAOS temporal logic conditions that refer to the past (such as trigger conditions). One can be more restrictive than necessary, e.g., by transforming conditions on some bounded-future states into conditions on the next state. In this case the operationalization loses its minimality property but keeps its completeness and consistency ones [26].

As experienced in the work on requirements and design patterns [21][14], pattern combination remains an open issue. Do patterns keep their effectiveness with respect to their associated non-functional requirements when they are combined? If not, how does on decide which one to choose? Does the order of application matter? Many such questions arise when using patterns.

Our current approach is purely refinement-based. This may not be sufficient in a number of situations where architectural features need to be propagated bottom-up, e.g., from middleware requirements. A complementary, dual approach based on abstraction patterns should be integrated to address this problem.

8. ACKNOWLEDGMENTS

This research was supported in part by NSF CISE grant CCR – 0306613 'Transforming requirements specifications into architectural descriptions', by the European Union (ERDF and ESF), and by the Walloon Region (DGTRE).

9. REFERENCES

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

[2] D. M. Berry, R. Kazman, and R. Wieringa, editors, *The Second International Software Requirements to Architectures Workshop* (STRAW'03). At ICSE'03.

[3] J. Bosch and P. Molin, "Software Architecture Design: Evaluation and Transformation", *Proc. IEEE Symp. On Engineering of Computer-Based Systems*, 1999.

[4] M. Brandozzi and D. E. Perry. "From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process", In [2], 107–113.

[5] M. Brandozzi and D. E. Perry, "Architecture Prescriptions for Dependable Systems", *Intl. Workshop on Architecting Dependable Systems*, ICSE'2002, Orlando FL, May 2002.

[6] J. Castro and J. Kramer, editors, T*he First International Workshop on From Software Requirements to Architectures* (STRAW'01). At ICSE'01.

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

[8] E. Ciapessoni, P. Mirandola, A. Coen-Porisini, D. Mandrioli, and A. Morzenti, "From Formal Models to Formally Based Methods: an Industrial Experience. *ACM Transactions on Software Engineering and Methodology* (TOSEM), Vol. 8 No. 1, January 1999, 79-113.

[9] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[10] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'4 – 4th ACM Symp. on Foundations of Software Engineering*, Oct. 1996.

[11] J. Dingel, D. Garlan, S. Jha, and D. Nokin, "Reasoning About Implicit Invocation", *Proc. FSE'6:* 6th *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Lake Buena Vista, November 1998.

[12] R. Dromey. "Architecture as an Emergent Property of Requirements Integration", In [2], pp. 77–84.

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

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

[15] D. Gross and E. Yu, "From Non-Functional Requirements to Design Through Patterns", *Requirements Engineering Journal* Vol. 6, 2001, 18-36.

[16] P. Grünbacher, A. Egyed, and N. Medvidovic, "Reconciling software requirements and architectures: The cbsp approach", *Proc. RE'01 - 5th Intl. Symp. Requirements Engineering,* 2001, pp. 202–211.

[17] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. "Relating Software Requirements and Architectures using Problem Frames", *Proc RE'02 -10th Int. Requirements Engineering Conference*, 2002.

[18] D. Jackson and K. Sullivan, "COM Revisited: Tool-Assisted Modelling of an Architectural Framework", *Proc. FSE'8:* 8th *ACM SIGSOFT Symp. on the Foundations of Software*

Engineering, San Diego, November 2000.

[19] D. Jani, D. Vanderveken, and D. E. Perry., *Experience Report: Deriving Architectural Specifications from KAOS Specifications*. December 2003. http://www.ece.utexas.edu/~perry/work/papers/R2A-ER.pdf

[20] D. Jani, D. Vanderveken, and D. E. Perry, "Deriving Architectural Specifications from KAOS Specifications: A Research Case Study", *European Workshop on Software Architecture*, Pisa IT, June 2005.

[21] S. Konrad and B.H.C. Cheng, "Requirements Patterns for Embedded Systems", *Proc RE'02 - 10th Int. Requirements Engineering Conference*, Essen (Germany), Sept. 2002.

[22] A. van Lamsweerde, R. Darimont, E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Transactions on Software Engineering,* Nov. 1998.

[23] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", *Proc. RE'01 - 5th Intl. Symp. Requirements Engineering*, August 2001.

[24] A. van Lamsweerde, "From System Goals to Software Architecture", In *Formal Methods for Software Architecture*, M. Bernardo & P. Inverardi (eds.), LNCS 2804, Springer-Verlag, 2003.

[25] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", in *Proc. ICSE'02: 24th Int. Conf. on Soft. Engineering*, May 2002.

[26] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", in *Proc. FSE'10:* 10th *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Charleston, November 2002.

[27] J. Magee, N Dulay, S. Eisenbach and J Kramer, "Specifying Distributed Software Architectures", *Proceedings ESEC'95 - 5th European Software Engineering Conference*, Sitges, LNCS 989, Springer-Verlag, Sept. 1995, 137-153.

[28] Z. Manna and A. Pnueli, The reactive behavior of reactive and concurrent system, Springer-Verlag, 1992.

[29] N. Medvidovic, P. Oreizy, J. Robbins, and R. Taylor, "Using Object-Oriented Typing to Support Architectural Design in the C2 Style", *Proc. FSE'4 - Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, October 1996.

[30] M. Moriconi, X. Qian, and R. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, Vol. 21 No. 4, Apr. 1995, 356-372.

[31] J. Mylopoulos et al. "Tropos - Requirements-Driven Development for Agent Software". http://www.troposproject.org/, 2004.

[32] B. Nuseibeh, "Weaving Together Requirements and Architecture", *IEEE Computer*, Vol. 34 No. 3, March 2001, 115-117.

[33] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, Vol. 17 No. 4, October 1992, 40-52.

[34] C. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde and H. Tran Van, "Early Verification and validation of Mission-Critical Systems", *Proc. FMICS'04*, Linz (Austria), Sept. 2004.

[35] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. "Architecture-driven Problem Decomposition", *Proc RE'4 - 12th Int. Requirements Engineering Conference*, 2004, 73–82.

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

[37] D. Vanderveken, *Deriving Architectural Descriptions From Goal-Oriented Requirements*. M.S. Thesis, University of Louvain, June 2004.