From System Goals to Software Architecture

Axel van Lamsweerde

Université catholique de Louvain, Département d'Ingénierie Informatique

B-1348 Louvain-la-Neuve (Belgium)

avl@info.ucl.ac.be

Abstract. Requirements and architecture are two essential inter-related products in the software lifecycle. Software architecture has long been recognized to have a profound impact on non-functional requirements about security, fault tolerance, performance, evolvability, and so forth. In spite of this, very few techniques are available to date for systematically building software architectures from functional and non-functional requirements so that such requirements are guaranteed by construction. The paper addresses this challenge and proposes a goal-oriented approach to architectural design based on the KAOS framework for modeling, specifying and analyzing requirements. After reviewing some global architectural decisions that are already involved in the requirements engineering process, we discuss our architecture derivation process. Software specifications are first derived from requirements. An abstract architectural draft is then derived from functional specifications. This draft is refined to meet domain-specific architectural constraints. The resulting architecture is then recursively refined to meet the various non-functional goals modelled and analyzed during the requirements engineering process.

1 Introduction

Requirements engineering (RE) is concerned with the elicitation of the goals to be achieved by the system envisioned (WHY issues), the operationalization of such goals into specifications of services and constraints (WHAT issues), and the assignment of responsibilities for the resulting requirements to agents such as humans, devices and software available or to be developed (WHO issues) [Lam00a].

Architectural design (AD) is concerned with the organization of the software-to-be into main components and interactions between them [Sha96, Bos00].

It has long been recognized that architectural design has a major impact on non-functional requirements about security, fault tolerance, performance, interoperability and maintainability [Per92, Sha96]. The problem of building an architecture which satisfies the software requirements is obviously central to software engineering. By and large, such building is however an ad hoc, largely informal and unsystematic process to date.

As a very first step, a rigorous architectural design process should rely on the use of precise descriptions of the software components and their interactions. Many architecture description languages (ADLs) have been proposed for this purpose, e.g., [All97, Gar97, Luc95, Mag95, Mor95, Med96]. An ADL captures the information required to guarantee desired properties related to the interaction of its components, as opposed to detailed design issues such as the choice of specific algorithms and data structures. ADLs provide support for explicitly modeling software components,

In *Formal Methods for Software Architectures*, M. Bernardo & P. Inverardi (eds), LNCS 2804, Springer-Verlag, 2003.

connectors, their configurations, and constraints on the components, connectors and configurations. One may thereby define a limited vocabulary of components and connectors, and rules by which they can be legally composed or legally interact. Some ADLs support architecture-level analysis to examine whether properties of interest are satisfied (e.g., absence of deadlocks). Other ADLs constrain component composition and run-time interactions so as to enforce the desired properties. ADL-based tools can then check conformance to rules of interaction and composition. Other tools can generate monitoring systems able to check at run-time whether the interaction rules are followed. Preliminary experience with ADLs suggest that architectural design based on such notations can be beneficial to the development, validation, maintenance, and reuse of software [Sha96]. For example, analysis at the architecture level has revealed anomalies and errors in a software integration framework for distributed simulation applications [All98]; new architecture-based integration tools have been successful at rapidly generating code for complex applications [Sha95].

Yet the key issue of constructing a software architecture that meets the elaborated requirements remains largely open. Very little work has been reported since Parnas' seminal work on heuristics for identifying components and dependencies among them [Par79]. In [Mor95], a formal framework is proposed in which correctness-preserving transformations can be applied to refine abstract architectures into concrete ones. Refinement patterns are also proposed there which are proved formally correct once for all and can be reused in matching situations. 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 [Bos99]. Gross and Yu show how the NFR goal-oriented qualitative framework from [Myl92, Chu00] can be used to document design patterns for selection during the architectural design process [Gro01]. In [Lam00a], an oversimplified procedure is just outlined by which components and dataflow connectors are derived first from functional requirements and then refined to meet non-functional goals through other types of refining connectors.

This paper presents some ongoing work on goal-oriented architecture derivation that goes far beyond our preliminary efforts. We put the following ideal (meta)requirements on our derivation process:

- the derivation should be systematic so as to provide active guidance to achitects,
- it should be incremental and allow for reasoning on partial models,
- it should lead to (at best) provably or (at least) arguably "correct" and "good" architectures –that is, meeting functional requirements and achieving non-functional ones,
- it should allow different architectural views to be highlighted, e.g., a security view, a fault tolerance view, etc.

At present stage, what we come up with is a systematic, goal-oriented process that partially intertwines requirements and architecture elaboration and at places allows for incremental, formal analysis of partial models through animation or checking against upstream, higher-level goal formulations.

Section 2 introduces some necessary background on goal-oriented model elaboration [Lam01]; it briefly recalls how software requirements can be incrementally derived from system goals and how high-level architectural choices are already made during that process. Section 3 shows how software specifications can be derived from requirements. The derivation of abstract dataflow architectures from functional software specifications is discussed in Section 4. The resulting architectural draft is refined first by imposing architectural styles on parts of it to meet domain-specific architectural constraints (Section 5). The next, iterative step then consists in refining this global, style-based architectural draft through local, pattern-directed refinement of components and connectors so as to meet the non-functional goals that emerged from the goal elaboration process (Section 6).

Throughout the exposition we will use the Meeting Scheduler benchmark as a running example [MOD]. The reader may refer to [Fea97] for a full problem statement.

2 Background

We introduce some basic concepts and terminology before recalling how goal, object, agent and operation models can be built systematically.

2.1 Goals, agents, objects and operations

A *goal* is a prescriptive statement of intent about some system (existing or to-be) whose satisfaction in general requires the cooperation of some of the agents forming that system. *Agents* are active components such as humans, devices, legacy software or software-to-be components that play some *role* towards goal satisfaction. Some agents thus define the software whereas the others define its environment; the word "system" refers to the software under consideration *and* its environment. Unlike goals, *domain properties* are descriptive statements about the environment –e.g., physical laws, organizational norms, etc.

Goals may refer to a wide variety of prescriptive assertions.

- *Functional goals* refer to services the system is expected to provide. For example, SatisfactionGoals are functional goals concerned with satisfying agent requests; InformationGoals are goals concerned with keeping agents informed about object states.
- *Non-functional goals* refer to quality of service, development objectives or architectural constraints.
  - *quality-of-service* goals capture application-specific concerns about safety, security, usability, performance, interoperability, accuracy of software information with respect to what it represents in the environment, etc.;
  - *development* goals refer to standard software quality criteria such as maintainability, reusability, etc.;
  - *architectural constraints* refer to domain-specific features of environment agents and relationships among them to be taken into account during architectural design –such as the distribution of human agents, organization

data or physical devices in the environment.

Goals are organized into AND/OR *refinement-abstraction structures* where higher-level goals are in general strategic, coarse-grained and involve multiple agents whereas lower-level goals are in general technical, fine-grained and involve less agents [Dar93, Dar96]. 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 an alternative set of refinements; this means that satisfying one of the refinements is a sufficient condition in the domain for satisfying the goal.

Goal refinement ends up when every subgoal is *realizable* by some individual agent assigned to it, that is, expressible in terms of conditions that are *monitorable* and *controllable* by the agent [Let02a]. A *requirement* is a realizable goal under responsibility of an agent in the software-to-be; an *expectation* is a realizable goal under responsibility of an agent in the environment (unlike requirements, expectations cannot be enforced by the software-to-be).

Goals prescribe *intended* behaviors; they can be formalized in a real-time linear temporal logic [Man92, Koy92, Dar93]. For example, one goal for a meeting scheduling system might assert that the date constraints of people expected to attend a meeting shall be known to the scheduler within *M* days after the meeting is requested:

Goal Achieve [ParticipantsConstraintsKnown]

FormalSpec $\forall$ m: Meeting, p: Participant

Requested (m) $\wedge$ Invited (p, m) $\wedge$ Scheduling (s, m) $\Rightarrow$ $\Diamond_{\leq Md}$ Knows (s, p.Constraints)

(Semi-formal keywords such as Achieve, Avoid, Maintain are used for lightweight reference to goals according to the temporal behavior pattern they prescribe.)

SoftGoals prescribe *preferred* behaviors; they are used to select preferred alternatives in an AND/OR goal refinement graph through qualitative reasoning [Myl92, Chu00]. SoftGoals can be refined but are in general hard to formalize.

The state of the system is defined by aggregation of the states of its objects. An *object* can be an entity, an association, an event or an agent (active object). Objects are characterized by attributes and domain properties (invariants). An *object model* is represented by a UML class diagram.

The *agent model* captures responsibility links between agents and goals together with monitoring/control links between agents and object attributes. The object attributes monitored and controlled by an agent define its *interface* to other agents [Let02a].

A goal assigned to some agent in the software-to-be is *operationalized* in functional services, called operations, to be performed by that agent. An *operation* is an input-output relation over objects; operation applications define state transitions. 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 condition* 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.

2.2 From system goals to software requirements

Operational software requirements are derived gradually from the underlying system goals. The derivation proceeds according to the following steps [Lam01].

- *Goal modeling:* A goal refinement graph is elaborated first by identifying relevant goals from input material (such as interview transcripts and available documents) –typically, by looking for intentional keywords in natural language statements and by asking *why* and *how* questions about such statements;
- *Object modeling:* UML classes, attributes and associations are derived systematically from goal specifications refering to them;
- *Agent modeling:* agents are identified, their monitoring/control capabilities are elicited from goal formulations, and alternative assignments of goals to agents are explored (alternative agent assignments define alternative system proposals and software/environment boundaries where more or less is automated);
- *Operationalization:* operations and their domain pre- and postconditions are identified from goal specifications; additional required pre-, post- and trigger conditions are derived so as to ensure the corresponding goals.

The above steps are ordered by data dependencies and, of course, intertwined. Each step is guided by heuristics and derivation patterns associated with specific tactics [Dar96, Let02a, Let02b]. Additional parallel steps of the method handle goal mining from scenarios [Lam98b], the management of conflicts between goals [Lam98a] and the management of obstacles to goal satisfaction [Lam00c], respectively.

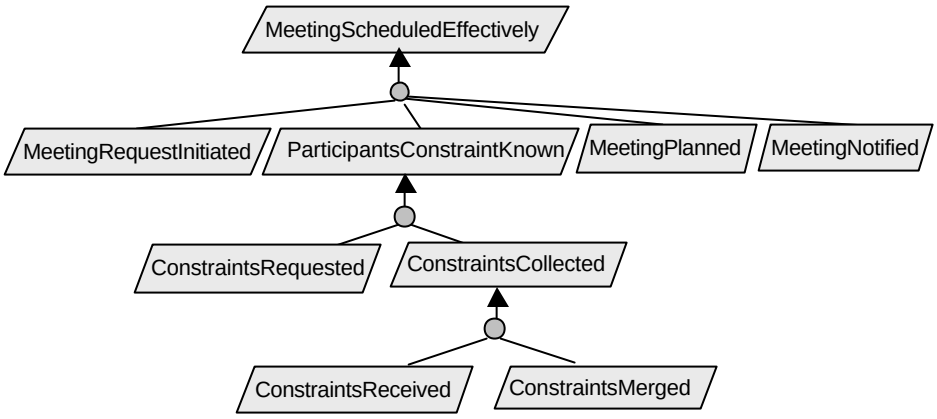

Fig. 1 – Portion of a goal refinement graph

Fig. 1 shows a portion of the goal model for our meeting scheduling system that

includes the goal Achieve [ParticipantsConstraintsKnown] formalized above. This goal was formally refined using the formal Refine-by-Milestone pattern twice (see Fig. 2). The goal Achieve [ConstraintsRequested] is formally operationalized into an operation RequestConstraintsToParticipants using the formal Bounded-Achieve pattern (see Fig. 3 where ***S*** denotes the *Since* temporal operator over past states [Man92]).

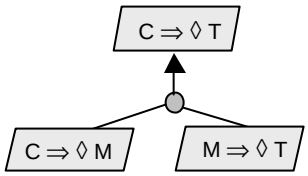

Fig. 2 – Refinement-by-milestone pattern

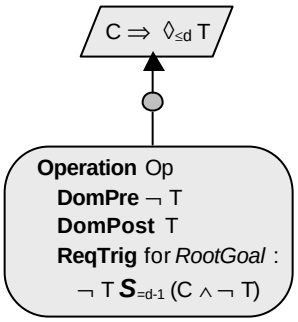

Fig. 3 – Bounded-Achieve operationalization pattern

2.3 On the invitable intertwining between requirements and architecture

It has long been recognized that specification and implementation are often intertwined in practice [Swa82]. More abstractly, problem and solution spaces are intertwined due to the recursive nature of problem solving –a problem is solved by specifying sub-problems and solving them. This observation has been remade recently in the context of requirements and architecture [Nus01].

In our framework, such intertwining appears at places where decisions have to be made among multiple alternatives being raised.

- A goal may be refined into several alternative AND-combinations of subgoals [Dar96];
- An obstacle obstructing a goal may be resolved through several alternative obstruction resolution tactics [Lam00c];
- A conflict among multiple goals may be resolved through several alternative conflict resolution tactics [Lam98];
- A "terminal" goal realizable by multiple agents may be assigned to several alternative candidate agents [Let02a]. When a software agent is being considered for assignment, there are alternative choices on the *granularity* of that agent –

from a fine-grained agent entirely dedicated to that goal to a global, coarse-grained "software-to-be" agent (see Section 4).

For each type of alternative, decisions have to be made which in the end will produce different architectures. A specific refinement, resolution or assignment is selected based on qualitative preferences dictated by positive contributions to high-priority softgoals [Myl92, Chu00, Lam00a] and/or resolution of other critical obstacles and conflicts [Lam00c, Lam98a]. Such early choices may have a global impact on the architecture.

Fig. 4 and 5 illustrate the point in the case of alternative refinements and assignments, respectively.

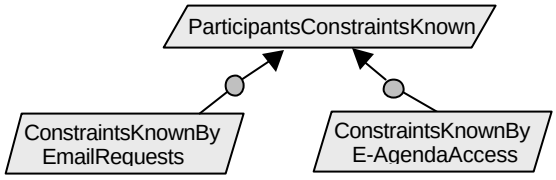

Fig. 4 – Alternative goal refinements

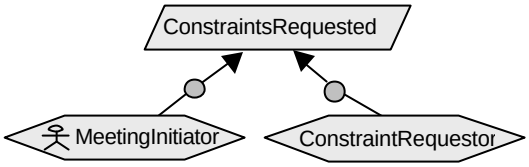

Fig. 5 – Alternative agent assignments

The global architecture of a meeting scheduler based on e-mail communication for getting participants constraints will be different from one based on the participant electronic agendas. There will be architectural differences between a version where meeting initiators are taking responsibility for handling constraint requests and a more automated version where a software component ConstraintRequestor will be responsible for this. [Lam00a] shows an example where two alternative goal refinements for a train control system lead to completely different architectures – from centralized to fully distributed.

3 From software requirements to software specifications

Requirements are formulated in terms of objects in the real world, in a vocabulary accessible to stakeholders [Jac95]; they capture required relations between objects in the environment that are monitored and controlled by the software, respectively [Par95].

Software specifications are formulated in terms of objects manipulated by the software, in a vocabulary accessible to programmers; they capture required relations between input and output software objects.

In our meeting scheduling example, consider the following requirement assignable to

some component of the meeting scheduler software:

Requirement Achieve [ConstraintsRequested]
FormalSpec $\forall$m: Meeting, p: Participant:
Requested (m) $\wedge$ Invited (p, m) $\Rightarrow$ $\Diamond_{\leq Rd}$ ConstrRequested (p)

In this formulation, the associations Requested, Invited and ConstrRequested correspond to phenomena that are observable in the environment. They need to be mapped to software input-ouput variables to produce, e.g., the following target software specification:

$\forall$m: MeetingClass, p: ParticipantClass
MeetRequest (m) $\wedge$ p **in** InviteeList (m) $\Rightarrow$ $\Diamond_{\leq Rd}$ ConstrReqSent (p)

Software specifications may be derived from requirements systematically as follows.

1. Translate all goals assigned to software agents into the vocabulary of the software-to-be by introduction of software input-output variables;
2. Map relevant elements of the (domain) object model to their images in the software's object model;
3. Introduce (non-functional) accuracy goals requiring the mapping to be consistent, that is, the state of software variables and database elements must accurately reflect the state of the corresponding monitored/controlled objects they represent [Dar93];
4. Introduce input/output agents to be responsible for such accuracy goals – typically, sensors, actuators or other environment agents.

For our above example, the accuracy goals will be

$\forall$m: Meeting, m': MeetingClass, p: Participant, p': ParticipantClass
Mapping (m, m') $\wedge$ Mapping (p, p') $\Rightarrow$
MeetRequest (m) $\Leftrightarrow$ Requested (m)
p' **in** InviteeList (m') $\Leftrightarrow$ Invited (p, m)
ConstrReqSent (p) $\Leftrightarrow$ ConstrRequested (p)

The first two equivalences will be assigned as expectations, e.g., to the MeetingInitiator agent (she has to include p' in the software input variable InviteeList iff that person is really among those expected to attend the meeting) whereas the third equivalence will be assigned as expectation, e.g., to the CommunicationInfrastructure agent.

Serious system failures are often caused by accuracy goal violations arising from environment agents not filling their expectations [Jac95, Lam00c]. If *Req* denotes the set of requirements assigned to software agents, *Exp* the set of expectations assigned to environment agents, *Dom* the set of domain properties, *Soft* the set of software specifications, *Acc* the set of accuracy goals, and *G* the set of goals under consideration, the following satisfaction relations must hold for every requirement *req* in *Req* and goal *g* in *G*:

Soft, Acc, Dom $\models$ req with Soft, Acc, Dom $\not\models$ **false**
Req, Exp, Dom $\models$ g with Req, Exp, Dom $\not\models$ **false**

4 From software specs to abstract dataflow architectures

From now on all the elaborated requirements and derived software specifications will be assumed to be non-conflicting as conflicts have been managed upstream in the requirements engineering process [Lam98a].

A first architectural draft is obtained from data dependencies among the software agents assigned to functional requirements. These agents become architectural components statically linked through dataflow connectors; there is no other "interaction" among the agents. In the transformation, the alternative of fine-grained components *C* associated with specific functional goals is preferred so as to address the non-functional softgoal *Maximize* [Cohesion (C)].

The procedure for deriving a dataflow architectural draft from our goal, agent and operation models is as follows.

1. For each functional goal assigned to the software-to-be, define one component regrouping a software agent dedicated to the goal together with the various operations operationalizing the goal and performed by the agent. The agent's interface is defined by the sets of variables the agent monitors and controls, respectively; such variables are derived from the goal assertion [Let02a] reformulated in terms of software variables according to the mapping defined in the previous step (see Section 3).
2. For each pair of components *C1* and *C2*, derive a dataflow connector from *C1* to *C2* labelled with variable *d* iff *d* is among *C1*'s controlled variables and *C2*'s monitored variables:

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

Fig. 6 shows a partial result of step 1 for a portion of the goal graph in Fig.1; Fig. 7 shows the dataflow architectural draft resulting from step 2.

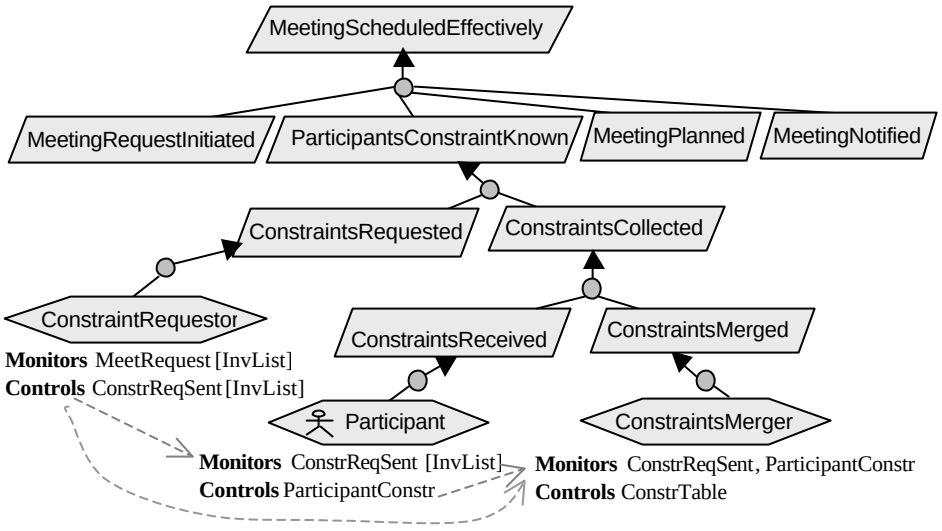

Fig. 6 – Assigned agents, their interfaces and data dependencies

The arrows in Fig. 7 denote dataflow connectors; they are labelled with corresponding data. Note that the ConstraintRequestor agent's interface in Fig. 6 was derived from the monitored and controlled conditions in the functional spec of the goal ConstraintRequested given in Section 3 and assigned to that agent.

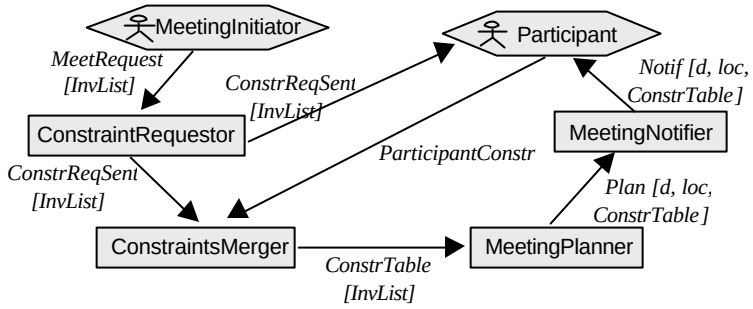

Fig. 7 – Derived dataflow architecture

In the dataflow architecture derived, each component is specified by the specification of the goal assigned to it together with the pre/trigger/post-conditions of the various operations operationalizing that goal.

When these specifications are formalized, our FAUST tools on top of the GRAIL environment [Dar98] can check at this abstract architecture level that the components together achieve higher-level goals from the goal graph, with counter-example scenarios being generated if this is not the case (we currently use bounded SAT solvers to do this). It can also generate state machines from the pre/trigger/post-condition specifications and animate them to vizualize whether the components behave as expected.

5 Style-based architecture refinement to meet architectural constraints

The initial abstract architecture obtained in Section 4 defines our refinement space. Before exploring alternative ways of refining components and connectors locally, this space may need to be globally constrained by architectural requirements. The latter 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 (see Section 2.1).

Our proposal here is to refine the dataflow architecture by imposing "suitable" architectural styles, that is, styles whose underlying (soft)goals match the architectural constraints. This requires such styles to be documented by applicability conditions (such as domain properties and the softgoals they are addressing [Gro01]) and effect conditions on the resulting architecture.

This step is currently fairly qualitative but can be made systematic through the use of transformation rules.

Fig. 8 shows a transformation rule for the introduction of the *event-based* style.

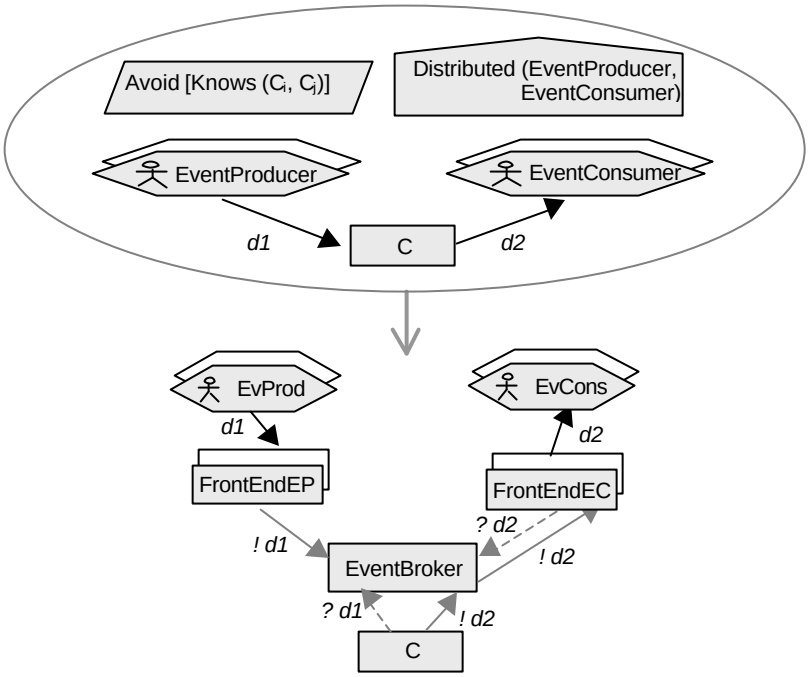

Fig. 8 – Introducing an event-based architectural style

The "home" notation is used there to denote a domain property. Standard arrows still denote dataflow connectors; a grey dashed arrow labelled by *?d* means that the source component *registers interest* to the target component for events corresponding to productions of *d*; a grey arrow labelled by *!d* means that the source component *notifies* the interested target component of events corresponding to productions of *d*. The latter events carry corresponding value for *d*.

Fig. 9 outlines a portion of the result of applying the style-based transformation in Fig. 8 to the abstract dataflow architecture in Fig. 7.

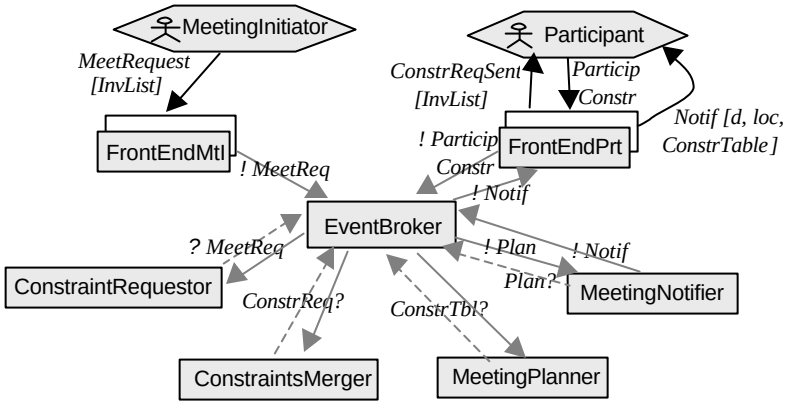

Fig. 9 – Style-based architecture to meet architectural constraints

Note that there are still data flowing through the gray event notification arrows as the events carry the corresponding data among their attributes. There is in fact a proof obligation that *refinements must preserve the properties of more abstract connectors and components*. In this case, an abstract dataflow channel between two components must be preserved either directly or indirectly through intermediate components (e.g., the EventBroker here).

6 Pattern-based architecture refinement to achieve non-functional requirements

Once an abstract dataflow architecture has been refined to meet architectural constraints it needs to be refined further in order to achieve the other types of non-functional goals, that is, quality-of-service goals and development goals. For example, the event broker in Fig. 9 should be split up into several brokers handling different kinds of events if the development goal *Maximize*[Cohesion(EventBroker)] is to be achieved. This is the next step of our derivation process.

Many quality-of-service goals impose constraints on component interaction. For example, security goals restrict interactions to limit information flows along channels; accuracy goals impose interactions to maintain a consistent state between related objects; usability requirements put constraints on information presentation and dialogs. Development goals such as *Minimize*[Coupling(C1,C2)] or InformationHidden(C1,C2) also impose specific constraints on the way the corresponding components may interact. On another hand, some non-functional goals impose constraints on single components only, e.g., *Maximize*[Cohesion(C)].

The next refinement step works on a more local basis than the previous one to "inject" quality-of-service and development goals within pairs of components (connector refinement) or single components (component refinement). The procedure is as follows. (We use NFG as an abbreviation for "quality-of-service or development goal".)

1. For each terminal NFG in the goal refinement graph *G*,
   - identify all specific connectors and components *G* may constrain;
   - instantiate *G* to those connectors and components (if necessary).
2. For each NFG-constrained connector or component, refine it to meet the instantiated NFGs associated with it; use *architectural refinement patterns* to drive the refinement as follows:
   - access a refinement pattern catalog where each pattern is a rewrite rule consisting of a source architectural fragment, a target architectural fragment refining that source, and a set of NFG goals achieved by the target,
   - select patterns whose source and NFG goals match the connector/components and the instantiated NFGs associated with them, respectively;
   - if there are several matching patterns, select a most preferred one based on NFG prioritization and tradeoff analysis (qualitative reasoning may be used to support this [Gro01]);

- apply the selected matching refinement pattern instantiated to the NFG-constrained connector or component to produce a new architectural fragment replacing the connector and connected components.

As a first example, consider the NoReadUpNoWriteDown pattern for confidentiality goals based on the Bell-LaPadula multi-level security model [Rie99]. Fig. 10 shows a formal representation of it. Note that the required postcondition of the refining component SecurityFilter is derived formally from the confidentiality goal specification using our formal operationalization patterns [Let00b].

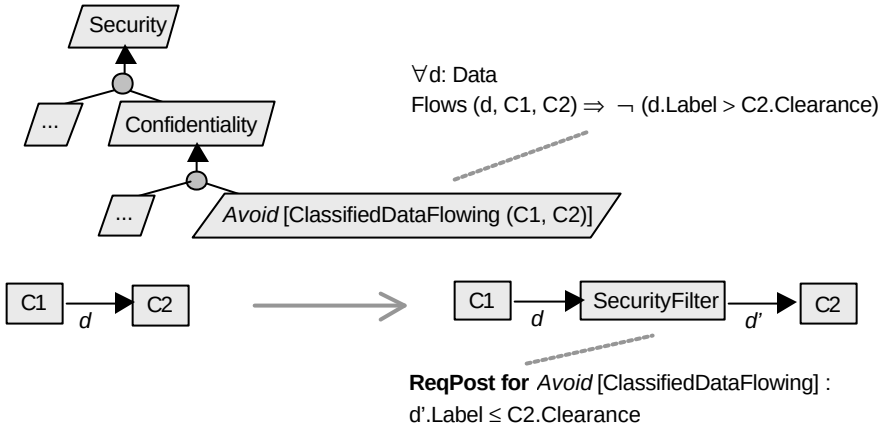

Fig. 10 – The *NoReadUpNoWriteDown* pattern for confidentiality goals

Let us now illustrate a pattern-based architectural refinement of the architectural draft for our meeting scheduling software obtained in Fig. 9.

Step 1 above results in localizing the impact of the confidentiality goal

Avoid [ParticipantConstraintsKnownToNonInitiatorParticipants]

from the full goal graph on the dataflow connector between the MeetingPlanner component and the MeetingNotifier component via the EventBroker component (see Figs. 7 and 9). In step 2 the NoReadUpNoWriteDown pattern is seen to be matching by considering two disclosure levels: one for meeting initiators, the other for normal participants.

The application of the intantiated pattern results in the introduction of a new component between the EventBroker and the MeetingNotifier:

ParticipantConstraintsFilter

that will ensure that participants constraints are filtered for normal participants from the data PlanningDetails attached to the event Notif transmitted from the MeetingPlanner to the MeetingNotifier via the EventBroker.

Fig. 11 and Fig. 12 suggest a sample of architectural patterns for quality-of-service and development goals, respectively. For our meeting scheduling software, the first pattern in Fig. 12 might be used to introduce a ConstraintsTable abstract data type component for use by the ConstraintsMerger and Planner components.

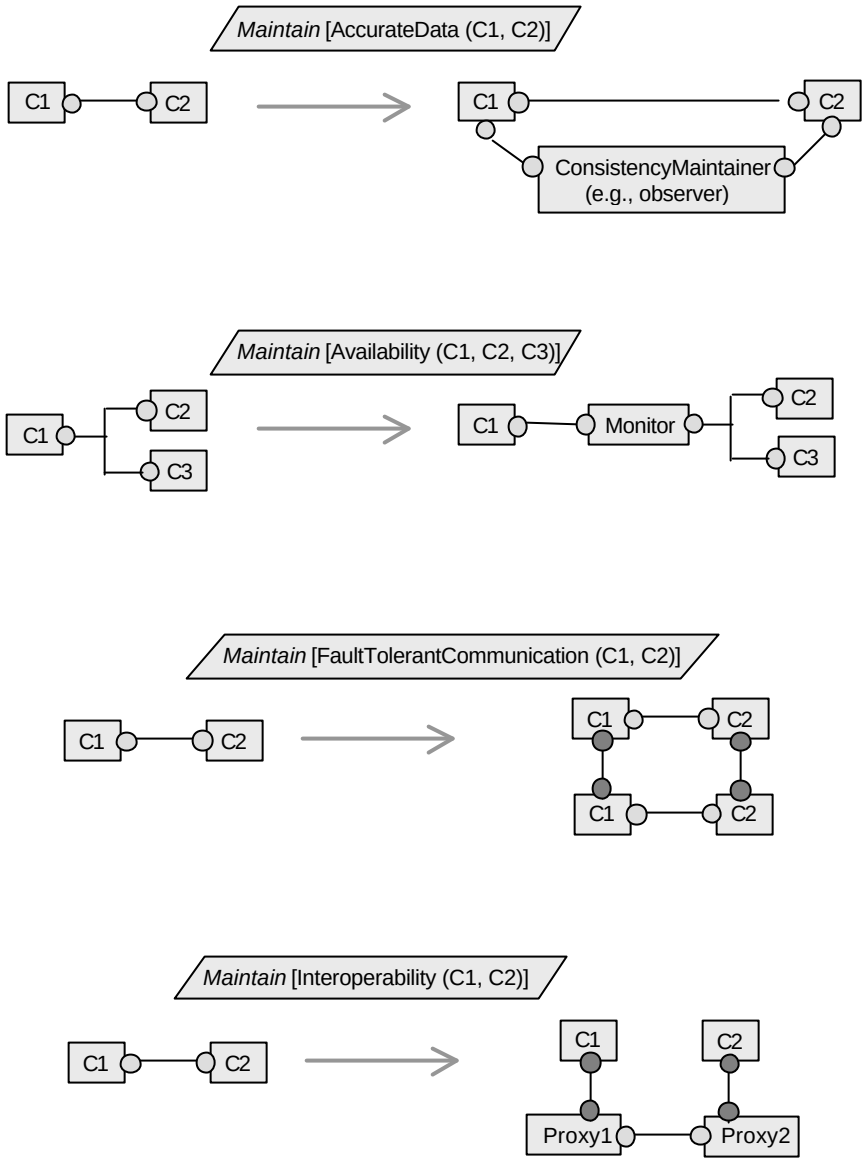

Fig. 11 – Architectural refinement patterns for quality-of-service goals

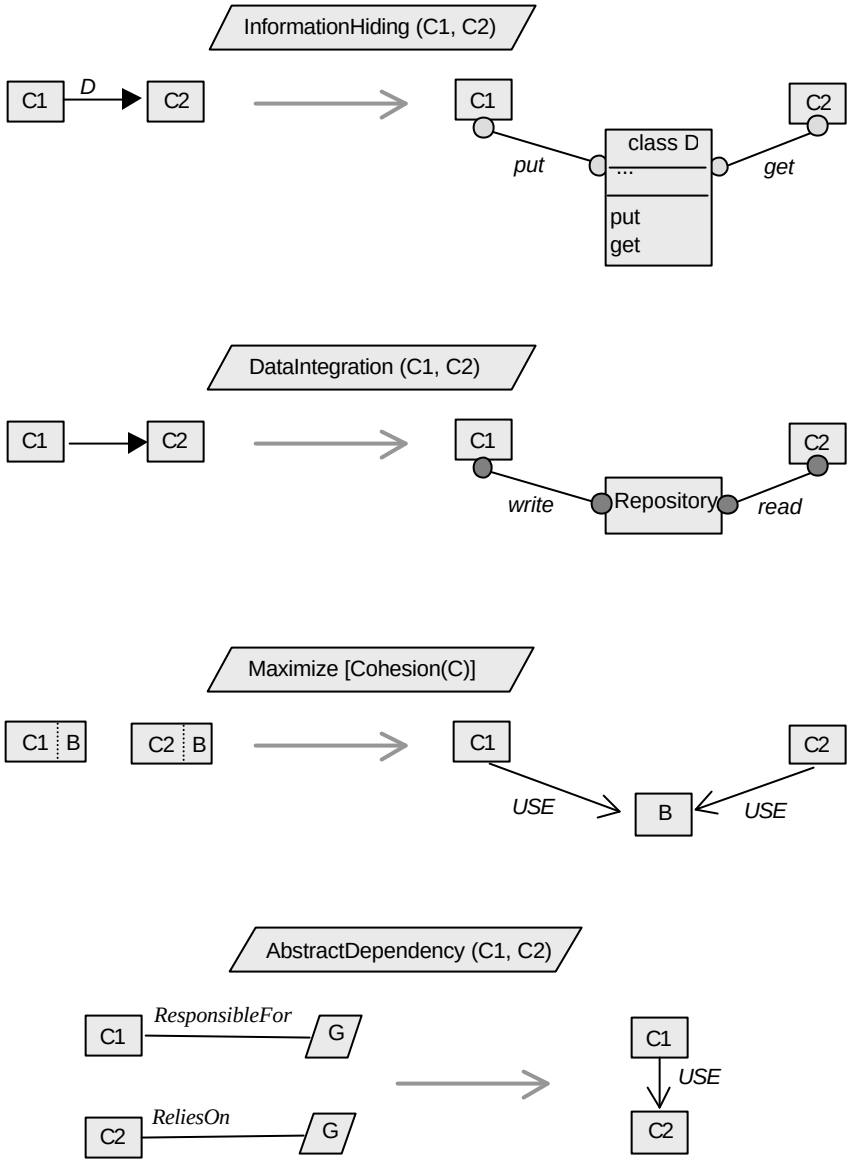

Fig. 12 – Architectural refinement patterns for development goals

7 Conclusion

We presented a systematic, incremental approach to deriving software architecture from system goals. The approach is grounded on the KAOS goal-oriented method for requirements engineering with the intent of exporting the virtues of goal orientation for constructive guidance of software architects in their design task. It mixes qualitative and formal reasoning towards the attainment of software architectures that meet both functional and non-functional requirements.

The architectural refinement of connectors and components is explicitly linked to the non-functional goals the refinement aims to achieve. This means that *architectural views* according to corresponding non-functional features (e.g., security views or fault tolerance views) are easily extracted through query systems such as the one provided by the GRAIL environment [Dar98].

This approach leaves a lot of questions open for further investigation though.

Up to what extent can the qualitative reasoning involved in architectural refinement be made more formal is an issue to be clarified if more sophisticated tool support is to be provided during the derivation process. In particular, the current style-based way of introducing architectural constraints leaves a lot of room for further improvement.

The proposed approach is purely refinement-based. This is clearly insufficient 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 might be worth investigating to address this problem.

In its current form, our approach does not reach the point where interaction protocols are detailed precisely. Our intent is to integrate previous, good-old-time results to formally derive such protocols, in particular through fixpoint computation of deadlock-free and starvation-free synchronizing schemes that achieve the goals [Lam79].

Acknowledgement. Insightful discussions on architectural refinement with R. Riemenschneider and D. Perry are gratefully acknowledged. Thanks are also due to members of the IFIP Working Group WG2.9 for feedback and criticism on earlier versions of this work.

References

[All97] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transctions on Software Engineering and Methodology*, Vol. 6 No. 3, July 1997, 213-249.

[All98] R. Allen, D. Garlan and J. Ivers, "Formal Modeling and Analysis of the HLA Component Integration Framework", *Proc. FSE-6 - 6th Intl Symposium on the Foundations of Software Engineering*, Lake Buena Vista, ACM, November 1998.

[Bos99] J. Bosch and P. Molin, "Software Architecture Design: Evaluation and Transformation", *Proc. IEEE Symp. On Engineering of Computer-Based Systems*, 1999.

[Bos00] J. Bosch, *Design and Use of Software Architectures – Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[Chu00] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, *Non-functional requirements in software engineering*. Kluwer Academic, Boston, 2000.

[Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[Dar96] R. Darimont and A. van Lamsweerde, "Formal Refinement Patterns for Goal-Driven Requirements Elaboration", *Proc. FSE'4 - Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

[Dar98] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde, "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering", *Proc. ICSE'98 - 20th Intl. Conf. on Software Engineering*, Kyoto, April 1998, vol. 2, 58-62. (Earlier and shorter version found in *Proc. ICSE'97 - 19th Intl. Conf. on Software Engineering*, Boston, May 1997, 612-613.

[Fea97] M. Feather, S. Fickas, A. Finkelstein, and A. van Lamsweerde, "Requirements and Specification Exemplars", Automated Software Engineering Vol. 4, 1997, 419-438.

[Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behaviour", *Proc. IWSSD'98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Gar97] D. Garlan, R. Monroe and D. Wile, "ACME: An Architecture Description Interchange Language", *Proceedings CASCON'97*, Toronto, Nov. 1997, 169-183.

[Gro01] D. Gross and E. Yu, "From Non-Functional Requirements to Design Through Patterns", *Requirements Engineering Journal* Vol. 6, 2001, 18-36.

[Jac95] M. Jackson, *Software Requirements & Specifications - A Lexicon of Practice, Principles and Pejudices*. ACM Press, Addison-Wesley, 1995.

[Koy92] R. Koymans, Specifying message passing and time-critical systems with temporal logic, *LNCS 651*, Springer-Verlag, 1992.

[Lam79] A. van Lamsweerde and M. Sintzoff, "Formal Derivation of Strongly Correct Concurrent Programs", *Acta Informatica* Vol. 12, 1979, 1-31.

[Lam95] A. van Lamsweerde, R. Darimont and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learned", *Proc. RE'95 - 2nd Int. Symp. on Requirements Engineering*, York, IEEE, 1995.

[Lam98a] A. van Lamsweerde, R. Darimont and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Trans. on Sofware. Engineering*, Special Issue on Inconsistency Management in Software Development, November 1998.

[Lam98b] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1089-1114.

[Lam00a] A. van Lamsweerde, "Requirements Engineering in the Year 00: A Research Perspective", Keynote paper, *Proc. ICSE'2000 - 22nd Intl. Conference on Software Engineering*, IEEE Press, June 2000.

[Lam00b] A. van Lamsweerde, "Formal Specification: a Roadmap". In *The Future of Software*

Engineering, A. Finkelstein (ed.), ACM Press, 2000.

[Lam00c] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, October 2000.

[Lam01] A. van Lamsweerde , "Goal-Oriented Requirements Engineering: A Guided Tour", *Invited Minitutorial, Proc. RE'01 - 5th Intl. Symp. Requirements Engineering*, Toronto, August 2001, pp. 249-263.

[Let02a] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *Proc. ICSE'02: 24th Intl. Conf. on Software Engineering*, Orlando, IEEE Computer Society Press, May 2002.

[Let02b] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", *Proc. FSE'10:* 10th *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Charleston, November 2002.

[Luc95] D. Luckham and J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, Vol. 21 No. 9, Sept. 1995, 717-734.

[Mag95] 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.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[Med96] 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.

[MOD] Model Problems for Software Architecture, http://www-2.cs.cmu.edu/People/ModProb/.

[Mor95] M. Moriconi, X. Qian, and R. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, Vol. 21 No. 4, Apr. 1995, 356-372.

[Myl92] Mylopoulos, J., Chung, L., Nixon, B., "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", *IEEE Trans. on Sofware. Engineering*, Vol. 18 No. 6, June 1992, pp. 483-497.

[Nus01] B. Nuseibeh, "Weaving Together Requirements and Architecture", IEEE Computer, Vol. 34 No. 3, March 2001, 115-117.

[Par79] D.L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering* SE-5 No. 2, March 1979.

[Par95] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", *Science of Computer Programming*, Vol. 25, 1995, 41-61.

[Per92] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, Vol. 17 No. 4, October 1992,40-52.

[Rie99] R.A. Riemenschneider, "Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures", *Proc. W1CSA1 - First IFIP Conference on Software Architecture*, San Antonio, February 1999.

[Rum99] J. Rumbaugh, I. Jacobson and G Booch, *The Unified Modeling Language Reference*

Manual. Addison-Wesley, Object Technology Series, 1999.

[Sha95] M. Shaw R. DeLine D. Klein, T. Ross, D. Young, and G. Zelesnick, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, Vol.21, No.4, April 1995, 314-335.

[Sha96] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[Sta95] The Standish Group, "Software Chaos", http:// www.standishgroup.com/chaos.html.

[Swa82] W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation", *Communications of the ACM*, Vol. 25 No. 7, July 1982, 438-440.

[Yue87] K. Yue, "What Does It Mean to Say that a Specification is Complete?", *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, IEEE, 1987.