CREWS Report 98-14

appeared in:
Proc. of CBISE'98 - CAiSE*98 Workshop on Component Based Information Systems Engineering, Pisa, Italy, June 8th-9th, 1998.

Layered Components

by

Luc Claes

Institut d'Informatique
Facultés Universitaires N.-D. de la Paix
rue Grandgagnage 21, Namur, Belgium
lclaes@info.fundp.ac.be

Layered Components *

Luc Claes

Computer Science Institute
University of Namur - Belgium
lclaes@info.fundp.ac.be

Abstract. *We describe a generic software environment intended to assist the construction of applications by the composition of reusable components. Two classes of components (and their related systems) are clearly distinguished: operational components (or components in the 'classical', technological sense) and descriptive components, encapsulating meta-data about, amongst others, their operational counterparts. Components are viewed as actors in a communicating entities system. Communications capabilities and needs are abstracted as a set of layers, each layer denoting a delimited conventions set and satisfying or dictating functional requirements.*

A case study describes how we are using - or intend to use - our models in real life applications. The selected case consists in the interconnection of two, otherwise incompatible, CASE tools.

1. Introduction

Every time our 'universal standardization' dreams are going away, we have to face that frustrating reality: software construction by components composition involves many heterogeneity-related issues: components technologies incompatibilities, frozen legacy applications, multiplicity of the interconnections and coding norms, not to mention the impediments raised by the various development 'culture gaps'. But maybe are we lucky in our misfortune: investigating the methods that could allow us to cope with that heterogeneity, could lead us to discover novel components design and/or selection rules favoring a better reusability.

In the next few pages, we roughly describe a software environment, allowing components to be assembled in a generic way by applying interoperability rules. Those rules assume a layered decomposition of inter-components communications. Furthermore, we evaluate the hypothesis suggesting that layered decompositions, of frequent use in data communications models (such as the ISO Open Systems Interconnection reference model [7]), are also applicable, at the cost of concepts generalizations, to the software components connectivity modeling.

Every communication implies the explicit or - more frequently - implicit sharing of a number of conventions between the communicating partners. By modeling that elementary principle and its associated rules, we expect to bring some useful components composition guidelines to the fore.

The software architecture described in this paper is currently tried out with the interconnection of various CASE tools. We will investigate our tentative experimentation plan.

2. Conceptual Background

2.1. Descriptive vs. Operational Components.

If it is quite usual, and natural, to conceptually isolate the active part of a component from its descriptive part - its 'meta-data' - we suggest here a more drastic approach, by granting the 'component' status to the meta-information as well.

* This work is founded by the European Community under ESPRIT Reactive Long Term Research 21.903 CREWS (Cooperative Requirements Engineering with Scenarios)

Components are divided in two classes: descriptive components (residing in a descriptive system) and operational components (residing in an operational system). Both systems are having their own ontology while interacting closely.

- *Operational* components are the actual actors, carrying out, by their conjugate actions, an expected 'run time' functionality. They are components in today's classical (technological) sense [12].

- *Descriptive* components encapsulate specifications. Those specifications are restricted to the communication-oriented framework described in the next paragraph. A descriptive component characterizes a *communicating entity*. A communicating entity is either an operational component or another individual intervening in the communication process while remaining somewhat 'out of control' in respect to our environment (Amongst others: operating systems services, legacy or otherwise existing applications, remote objects, hardware devices, operating system services and, more generally, the communication environment.)

The descriptive/operational separation allows our models to capture information about all relevant communication actors. Meta-data being made accessible as individual components as well, could be stored locally or remotely in components catalogs or repositories. A componentized software configuration needs to be built and validated at the 'descriptive' level before being instantiated and activated at the operational level.

2.2. (Descriptive) Components Layering.

Descriptive components are organized as a set of layers, reflecting the communication structure of a communicating entity.

Each layer has a set of properties, of which we selected:

- The **Protocol**. In a much broader sense than in data transmissions, the protocol is defined here as a *delimited set of conventions*. It could be a programming interface, a classical communication protocol, a well-defined set of assumptions made about the environment, etc. A shared protocol is a necessary condition for a communication to take place. The protocols are only identified, localized in a taxonomy allowing specialization / generalization relationships. The model does not capture the precise protocol semantics.
- **Polarity**. Does the communicating partners, at a given layer, have an identical status (= neutral polarity) or are they 'asymmetrical', like, for example, in client / server or master / slave relationships? Arbitrarily, the 'originating' actor is given a positive polarity, while the 'answering' one has a negative polarity.
- **Multiplicity** expressed as an interval [m, M]. That property synthesizes two sub-properties: (i) does this layer act as a requester (m > 0) or as a provider (m = 0 or undefined) of the service identified by the protocol (ii) does this layer support multicasting (M > 1 or undefined), by being able to serve multiple requesters simultaneously.
- Beyond those intra-layer properties, an inter-layer relationship expresses dependencies of the kind 'Layer x **uses** the services provided by Layer y' defining a directed acyclic graph, of which the classical 'protocol stack' is a particular case.

The properties we have described participate in the elaboration of a rules system, constituted of a set of functional requirements that each component layer will in turn satisfy or require. Ideally, all conditions necessary to achieve the components interoperations should be expressed, even the most implicit one, starting from the lower level aspects (operating system services, media access, processors, virtual machines...) up to the communication semantic aspects.

3. Case Study

3.1. Problem Statement.

Let us introduce an experimentation plan where the actors are various CASE tools, and of which the central element is a graphical/textual editor allowing users to express specifications using the formal requirements engineering language *Albert-II* [2]. Our objective is to set up interconnections between that software application and an array of somewhat related tools, like specification animators, theorem provers, traceability tools, etc.

Concretely, we are currently implementing a 'bridge' between the *Albert* Editor and the requirements engineering environment *Pro-Art* [10]. Pro-Art is used in order to capture trace information throughout the Albert-II specification process.

We consider successive transitions starting from a straightforward static components assembling and ending with a more ambitious, almost automatic and optimized solution, following a 4-steps schema.

Let us first examine our problem given information by sketching the initial descriptive components. It is worth to mention that many simplifications have been introduced: for example, the user-interface is not modeled, protocols are too generic and most low-level services are oversimplified.

In order to describe the various layers, we use the graphical syntax of Figure 1.

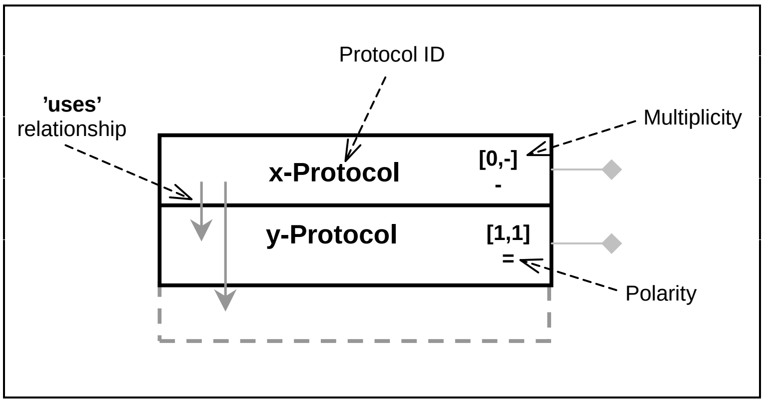

Figure 1 Graphical Syntax

We are considering the central piece of software (the *Albert* editor) only with respect to its inter-applications connectivity. That connectivity was implemented by setting up a generic communication mechanism, offering a controlled access to all Albert editor's relevant/useful methods, properties and events. We expect that no restrictive assumption be made in this interface definition concerning the potential software applications that could potentially use it.

As the editor did already represent a respectable amount of code, we were implicitly led to a technological solution considering that investment (C++, MFCs, Win32,...). We were also almost inevitably guided to implement a COM connectivity (Component Object Model: the foundation of Microsoft's OLE and ActiveX [1]), materialized by COM-compliant interfaces (custom, 'early bound' interfaces and automation, 'late bound' interfaces). However, obviously, our models are independent from the components technology.

Our - now 'open' - editor could be represented, as the set of communication layers of Figure 2:

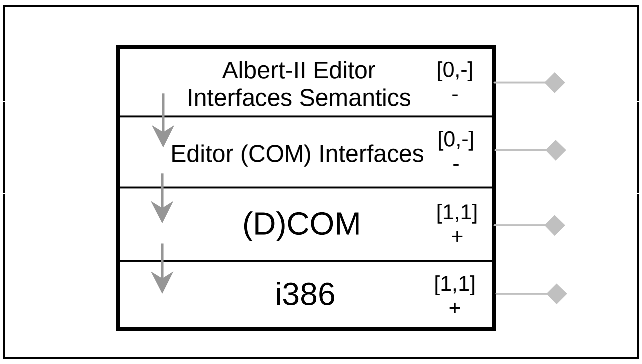

Figure 2 - Albert-II Editor Descriptive Component

The different communication levels, each dictating or satisfying a necessary interoperability condition, could be roughly described as follows (Table 1), starting from the most 'semantic' and ending with the most 'physical' level.

Layer	Description	Specified/described by
Albert-II Interface Semantics	Albert-II language semantics (subset)	Intuitive and formal semantics ; metamodel
Editor COM Interfaces	Albert Editor custom COM interfaces	IDL specification; Interface dynamic behavior formal or intuitive specification.
DCOM	Distributed Component Object Model operating environment	Microsoft documents
i386	Processor: data format, instructions set	Intel documents

Table 1 - Albert-II Editor Layers Description

To illustrate the properties values:

- A layer implements the *Editor (COM) Interfaces 'protocol'* with a negative polarity, meaning it acts as a 'server' for potential 'clients': it *offers* a COM interfaces implementation. The [0,-] indicates that no communication partner is required, but that an indefinite number of such partners could be 'served'.
- Another layer indicates that the software requires the presence of a DCOM environment. Here, it acts as a 'client' (positive polarity) and the service is required ([1,1] multiplicity).

The *'PRO-ART'* software we intend to connect to the Albert-II language editor is described, from our communication point of view, as follows (Figure 3). Beyond their actual application-level semantics, Pro-Art services are exposed through a (proprietary) object-model protocol, using a classical TCP/IP transport.

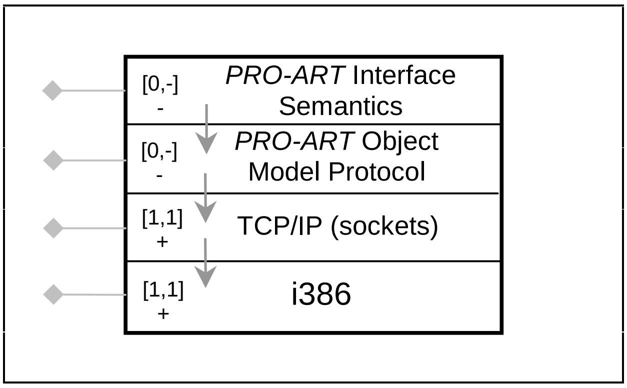

Figure 3 *PRO-ART* Descriptive Component

3.2. Introducing Bridgelets: Protocol(s) Converter Components.

Interoperability between the PRO-ART software and the Albert editor is obtained by creating one or more (here: 2) intermediate components implementing the necessary communication adaptations, up to the higher levels of 'incompatibility' (here: semantic).

The first component - the PRO-ART/COM *bridgelet* (Figure 4) - is designed in order to offer a reusable front-end to PRO-ART, masking the complex proprietary object model protocol and hiding the TCP/IP low-level housekeeping as well. Note that this bridgelet has no semantic-level knowledge. It acts only as a 'syntactical' bridge, carrying out straightforward translations.

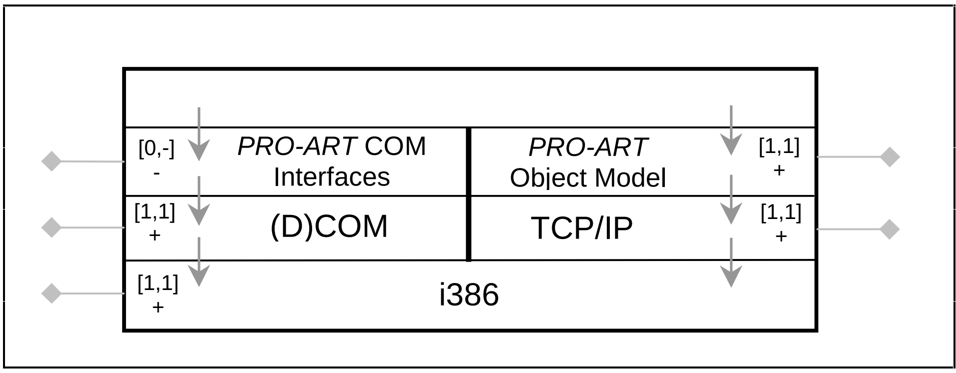

Figure 4 - PRO-ART / COM 'bridgelet' (Descriptive) Component

A second component - the *Albert* editor / *PRO-ART* bridgelet (Figure 5) - implements the actual high-level bridging. Noteworthy, the low-level software technology is here somewhat different: Java is used as implementation language, instead of a machine-dependant C++ compiled solution.

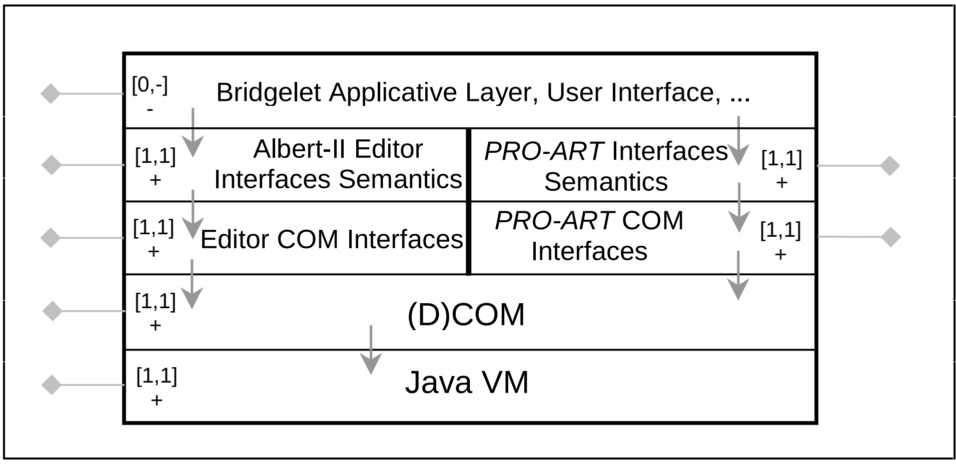

Figure 5 - *Albert-II* / *PRO-ART* 'Bridgelet' Descriptive Component

The overall configuration, satisfying all functional requirements could be described as follows (Figure 6).

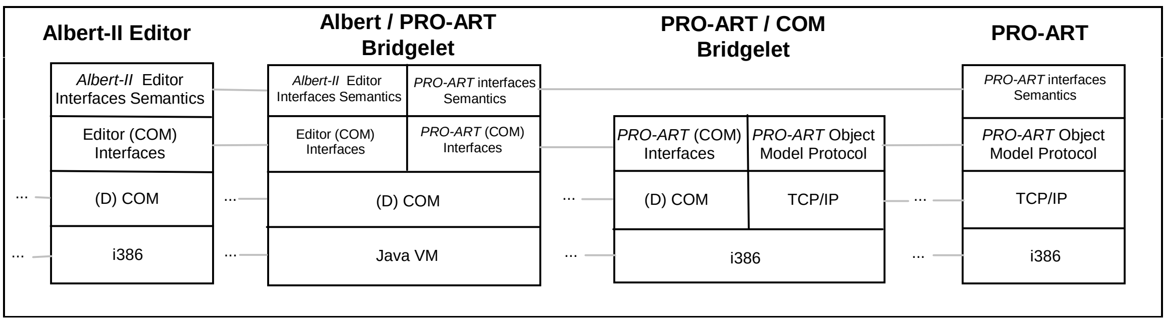

Figure 6 - Overall (Descriptive) Configuration

3.3. Experimentation plan

Let us describe a 4-phases experimentation plan, reflecting a bottom-up process.

Starting from a purely operational architecture, we gradually enrich the descriptive system. A components composition framework is progressively constructed.

3.3.1. Phase 1. Specific, 'Manual' Components Composition.

During phase 1, existing applications and operational components are assembled statically at design time. The Albert editor is modified in order to act as a container for 'well-known' components. Those components are instantiated and interconnected by specific code.

3.3.2. Phase 2. Generic, 'Manual' Components Composition.

Phase 2 targets to establish a solid conceptual foundation for the next phases. The graphical, intuitive syntax of Figure 1 is extended and formalized:

- The descriptive and operational systems ontologies are currently modeled using the *Telos* notation [8]. Telos is a conceptual modeling language in the family of entity-relationship models, supporting unlimited classification levels (meta-classes) and treating attributes as full-fledged objects.
- The models inferential aspects and integrity constraints are tentatively expressed using *Object-Z*, an object-oriented extension to the specification language *Z* [3].

A simplified components composition framework is set up, allowing components configurations to be constructed and validated at the descriptive level. Simple configurations activation, with adequate operational components instantiations is supported.

3.3.3. Phase 3. Generic, Assisted Components Composition.

A 'Phase-3' framework is able to generate components configurations (semi-) automatically, on the basis of (i) requirements expressed by the user as a given set of descriptive components (ii) a formalized knowledge about the operating environment and available resources (that is, catalogs of locally or remotely available components). Criteria taken into account at this stage are only targeting basic components interoperability: communication necessary conditions, functional requirements have to be satisfied.

Regarding our case study, this means that, given an initial, incomplete configuration containing two descriptive components ('Albert Editor' and 'PRO-ART'), the framework has to infer a complete, valid configuration by fetching components satisfying all constraint, (for example the 'Albert/PRO-ART' and 'PRO-ART/COM' bridgelets, but other solutions could be found) in components catalogs.

An example of such configuration automation, somewhat restricted, is described by [4].

3.3.4. Phase 4. Generic, Assisted, Optimized Components Composition.

The 'Phase-3' framework could generate, for a given requirements set, several valid configurations (i.e. satisfying all constraints). During this fourth phase, we intend to integrate the ability to consider some non-functional requirements, in order to select not only a 'valid' configuration, but also the 'best' one. Interesting criterions would be, amongst others, connections cost, performance, security, etc.

4. Conclusions

We described a 'components world', with a partial map like (Figure 7).

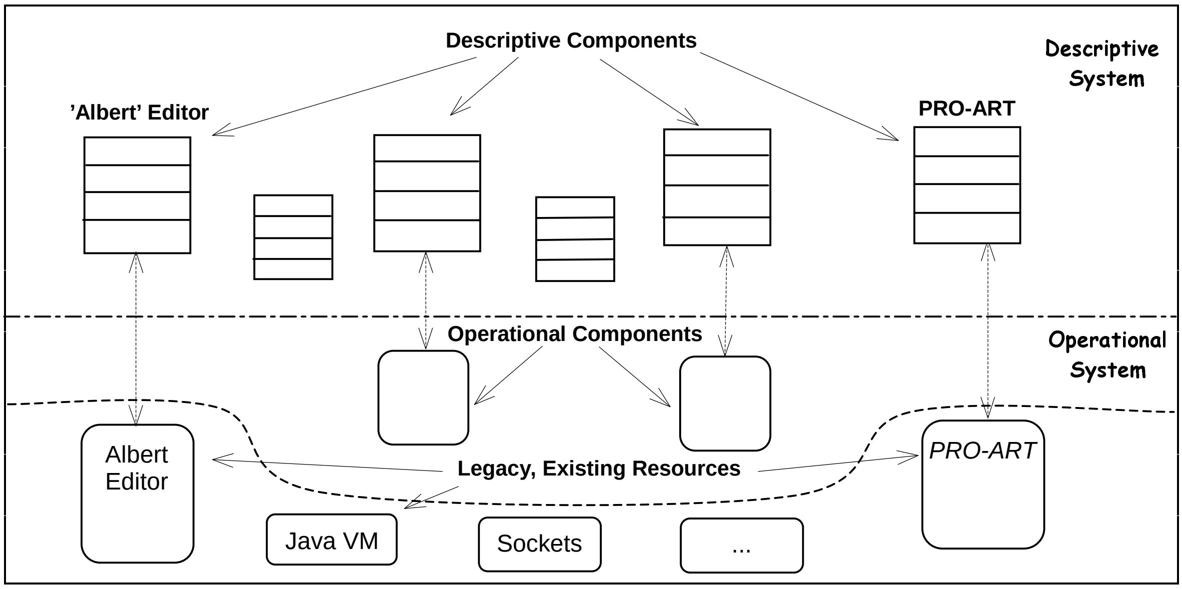

Figure 7

However, our map contains many white areas, of which we could mention:

- In order to break down the components specification into layers, we need an accurate set of criteria, like 'substitutability'. Nowadays, that decomposition lacks precision, particularly at the higher, semantic levels.
- Some fundamental dynamic aspects are not taken into account: initialization, activation, finalization, components sharing among multiple configurations, etc.
- Inter-layer relationships deserve a more precise modeling. By simply indicating that 'Layer A **uses** Layer B', the information loss is often unacceptable.

Acknowledgements This work is founded by the European Community under ESPRIT Reactive Long Term Research 21.903 CREWS (Cooperative Requirements Engineering with Scenarios). The author would like to thank Eric Dubois and Patrick Heymans for their comments on an earlier version of this paper.

References

[1] Kraig Brockschmidt, *Inside OLE*, 2nd edition, Microsoft Press, 1995.

[2] Philippe Du Bois, *The Albert-II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis*, PhD Thesis, University of Namur (Belgium), September 1995.

[3] Roger Duke, Paul King, Gordon Rose, Graeme Smith, *"The Object-Z Specification Language Version 1"*, Software Verification Research Centre Technical Report 91-1, University of Queensland, May 1991.

[4] David Hovel, *'Using Prolog in Windows NT Network Configuration'*, presented at *Practical Applications in Prolog'95*, Paris, April 1995.

[5] Michael R. Genesereth, Narinder P. Singh, Mustafa A. Syed, *"A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation"*, 3rd CIKM Workshop on Intelligent Information Agents, Gaithersburg, Maryland, December 1994.

[6] Sandra Heiler, Renée J. Miller, Vincent Ventrone, "*Using Metadata to Address Problems of Semantic Interoperability in Large Objects Systems*", Proceedings of the First IEEE Metadata Conference, Silver Spring, April 1996.

[7] International Organization for Standardization, *"OSI Basic Reference Model"*, ISO 7498-1, 1992. (Also known as ITU-T X200)

[8] John Mylopoulos, Alex Borgida, Matthias Jarke, M. Koubarakis, *"Telos - a language for representing knowledge about information systems"*, in ACM Transactions on Information Systems, 8, 4, 1990, pp. 325-362.

[9] Xavier Pintado, *"Gluons and the Cooperation between Software Components"*, in O.Nierstrasz & D. Tsichritzis *Object-Oriented Software Composition*, Prentice-Hall, 1995.

[10] Klaus Pohl, *Process-Centered Requirements Engineering*, Research Studies Press Ltd., 1996.

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

[12] John Udell, “Componentware”, *Byte*, May 1994, pp. 46-56.