A Design Framework for Distributed Data Acquisition and Triggering Systems in High Energy Physics Experiments

Shebli Anvar*, François Terrier†
* DAPNIA/SEI – CEA Saclay – F-91191 Gif, France, S.Anvar@cea.fr
† LETI/DEIN – CEA Saclay – F-91191 Gif, France, F.Terrier@cea.fr

Abstract

We present the first conceptual results in the design of an object-oriented framework that tackles recurrent problems encountered when developing acquisition and triggering systems for high-energy physics experiments. These problems include software/hardware frontier definition and the impact of both intrinsic and performance-related distribution on software development. Based on the UML (Unified Modeling Language) extension mechanisms and a real-time CORBA (Common Object Request Broker Architecture) implementation, the framework aims at providing 1) high-level concepts for flexible hardware-software separation, and 2) simple mechanisms for transparently deriving distribution automatically from system-level definitions. The framework will therefore implement design patterns that result from our experience of HEP (High Energy Physics) TDAQ (Trigger and Data Acquisition) system development.

I. INTRODUCTION

The digital systems developed for data acquisition and real-time processing in HEP experiments keep getting bigger and globally more complex [1], [2]. The constraints on such systems are more stringent that ever both in terms of performance and robustness. To be able to cope with the design and development challenges that such growing complexity and size entail, the engineers of the HEP community have to fathom new methods and tools that make these future systems feasible, cost-effective and maintainable over many years.

Computer science experience and applied research [3]- [8] show that major solutions to system design rationalization always involve the abstraction of processes and patterns specific to the application domain and their implementation as reusable components inside a domain-specific framework. We have therefore started a reflection leading us to elaborate on a design method specifically adapted to data acquisition and trigger systems such as those developed in HEP-like experiments. We have begun implementing the results of our reflection in the form of a design and development framework, eventually including reusable design architectures. In this paper, we discuss a number of concepts and ideas related to recurrent problems encountered in the design of Trigger/DAQ systems.

In section II we discuss the necessity of a separation of concerns between distribution issues and functional issues during the design process, due to the need for multiple distribution schemes, flexibility in the definition of hardware/software frontier and difference in design impacts of performance-related distribution and intrinsic distribution. In section III, we briefly justify our choice of the UML notation and, using a toy example, we present two classes of UML diagrams respectively devoted to functional and deployment specification. We also point out some UML extensions needed for our purposes. In section IV, we present a scheme for the automation of distribution-specific patterns through the analysis of functional and deployment specifications. Section V is devoted to a discussion on the choice of development and distribution technologies for a TDAQ framework in the context of HEP experiments, especially taking into account maintainability and evolution in projects with lifetimes that span over many years. We conclude the paper in section VI by summarizing the showing the benefits of these first ideas for a TDAQ design framework and presenting future research plans on the subject.

II. SEPARATING FUNCTIONAL DESIGN AND SYSTEM DEPLOYMENT

A. Multiple Distribution Schemes During Design and Development

HEP TDAQ systems are large-scale, complex systems that necessitate a progressive and incremental design process. Basically, we have physics detectors whose topology and qualities are essentially determined by the scientific principles on which the experiment is based. They produce signals that are digitized by the front-end electronics. The function of the TDAQ system is to process the data produced by the front-end electronics and store the results for further scientific analysis. Therefore, the TDAQ subsystem designers are basically given the geometrical organization of an input data-flow together with a quantitative estimation of its extent. From that, they must design a subsystem that carries out the required processing and/or acquisition.

The design process goes through a general specification phase in which the final distribution scheme is fleshed out in principle by specifying the processing nodes (participants) and their interconnections (topology). The actual design and development of the subsystem then consists in a considerable number of *successive* hardware/software developments, design refinements, tests and optimizations. We might say that the final system will progressively grow from a seminal simple system through successive complexifications. This is an *iterative* process in which each cycle will produce an

intermediary distribution scheme. For instance, we might begin with a purely functional program running on a single computer and then deploy the program on a small multi-node network, then on a heterogeneous network involving more specific processor types needed in different parts of the system, etc. until the system reaches its final size and complexity. Each cycle might involve redeployments, hardware/software design and development, software porting, and actual tests and measurements of the corresponding setup that lead to further optimizations. As a consequence, before the final distribution scheme is actually implemented, the TDAQ designers have to implement numerous smaller, tentative schemes. During this iterative process, the functional design of the system is likely to evolve at a different (slower) pace than the distribution architecture. Consequently, in order to minimize re-designs and code re-development, a design framework for TDAQ systems should provide for means of *separating* these two aspects, so that evolutions of the distribution scheme entail minimal modifications in the system's functional architecture.

B. The Hardware/Software Frontier Problem

High-energy physics experiments always depend on custom-made front-end detectors that need specific electronic systems for read-out and acquisition. Behind the physics detectors – such as photomultipliers, wire chambers or CCDs – specific hardware is necessary to give form to detector signals, digitize and do some real-time processing on the resulting data flow. However, specific hardware must stop *as soon as possible* in the acquisition chain in order to pass on the data to flexible (i.e. software) subsystems running over COTS (Commercial Off-The-Shelf) electronics and a standard OS (Operating System). The necessity of such flexibility stems from the fact that time scales of both the design and the exploitation of HEP physics experiments tend to extend over many years, which calls for possibilities of modifying designs even during the development cycle in order to keep up with technological advances and evolving standards. Consequently, the *frontier between hardware and software* components inside a subsystem *should be decided for as late as possible* in the design process. This calls for techniques that allow the *design* process to go on without having to specify which components will be hardware and which will be software. The results of this design process must be then *deployed* over a specific hardware configuration. The core of any framework solution tackling this problem will therefore be a *separation of concern* between functional system design and deployment design. In other terms, the hardware/software frontier determination can actually be treated as a *deployment* problem: if an object is deployed as a hardware component, then no code is generated for it (or firmware code such as VHDL code), and the designer must only define a precise interface for communication between code and hardware.

C. Intrinsic and Performance Distribution

The data acquisition systems and their associated trigger systems are usually *distributed by nature*, because detectors are spread out — sometimes over great distances. In collider experiments, for instance, tens of thousands (if not millions) of electronic channels must be digitized, data-formatted, processed and stored [9]; in some astroparticle experiments such as ANTARES [10], detector nodes are spread over volumes of millions of cubic meters. We call this kind of distributed feature "intrinsic distribution."

In addition to that, the data flows in HEP TDAQ systems are often considerable and consequently call for processing power that cannot be provided by single machines. In such systems, intrinsic distribution is therefore accompanied by "performance distribution," that is, distribution that arises due to the scaling up of processing power through parallelization, such as in computing farms. In short, whether intrinsically or for the sake of processing power, HEP TDAQ systems are most often massively distributed systems.

1) Design Impacts of Intrinsic Distribution

The distinction between performance distribution and intrinsic distribution has non-negligible impact on system design. Intrinsic distribution stems from the spreading out of processing nodes that are close to the front-end electronics. These nodes contain specific electronics and are often embedded and hard –if not impossible– to access. As a consequence, they may be highly evolving during design but after that, they are not liable to evolve much. Intrinsic distribution in HEP TDAQ systems is therefore quite static throughout the detector's lifetime; it is also sensitive to failures as the lack of certain detector points may violate the principle of the experiment (for instance, the failure of key regions in a track reconstruction detector might make it impossible to reconstruct any particle track). That is why the intrinsically distributed part of a TDAQ system often calls for more robust designs.

2) Design Impacts of Performance Distribution

Performance distribution, on the other hand, is mostly found in the form of processor farms that are accessible and easily upgradeable. Performance-distributed subsystems are therefore likely to have a fast evolution rate in order to keep up with technological advances. Also, their scalability makes them less sensitive to failures, as processor failures do result in a degradation of performance but are less liable to cause a complete breakdown.

3) Impact on Design Framework

These qualitative differences in design requirements between intrinsically distributed and performance distributed subsystems imply that any design framework for HEP TDAQ systems must allow for such a distinction in order for the

design process to be able to apply specific procedures for each.

III. FUNCTIONAL AND DEPLOYMENT DIAGRAMS

A. Why Use the UML?

An HEP TDAQ design framework that enforces the separation of concerns discussed in subsections A and B should clearly distinguish two classes of specifications, one for functional and one for deployment concerns. The UML notation provides for diagrams that are quite adapted for that purpose. It also provides for standard extension mechanisms for specializing the notation and adapting it the specific domains (such as HEP TDAQs in our case). Moreover, the UML is today universally recognized as the definitive standard for object systems modeling [12][13] and all modern methods and software development frameworks are UML-based. We have therefore decided to base our own TDAQ design framework on UML notation and modeling. This will ensure maximum compatibility with COTS development frameworks and prevent us from heavily relying on proprietary languages and notations.

B. The Two Classes of UML Diagrams

The twofold system specification in our TDAQ framework would rely on two classes of UML diagrams: the first class would be devoted to functional design and would include all the UML static and dynamic specification diagrams such as class, collaboration, sequence, activity or state transition diagrams. The second class of diagrams would be essentially based on the UML deployment and/or component diagrams. Ideally, the system designer should be able to: 1) define the system as *one* program in the form of a set of interacting functional objects that implement the processing algorithms that the system is expected to perform and 2) specify *many* deployment schemes that represent as many ways of running the program on different network topologies.

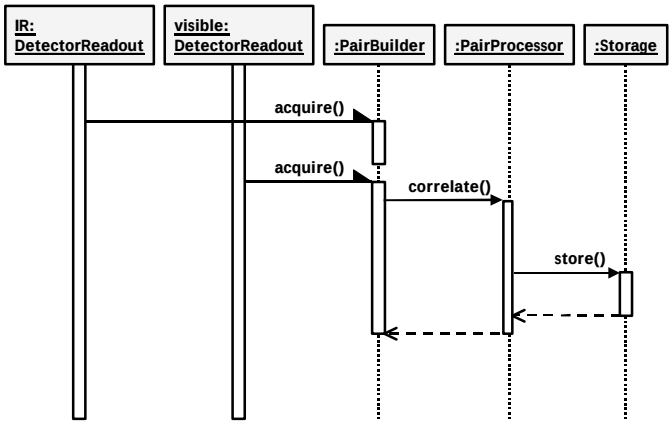

Figure 1: Sequence Diagram of SAMSA System.

C. Diagrams for a Simple Example

Let us consider, for instance, a Simplistic Astrophysical Multi-Spectral Analysis (SAMSA) system that processes images coming from two detectors attached to a telescope. Detector 1 is sensitive to infrared light and detector 2 to visible light. The system must first bundle pairs of images, then find correlation patterns between the infrared and visible images and then store the images and the results in a compressed format.

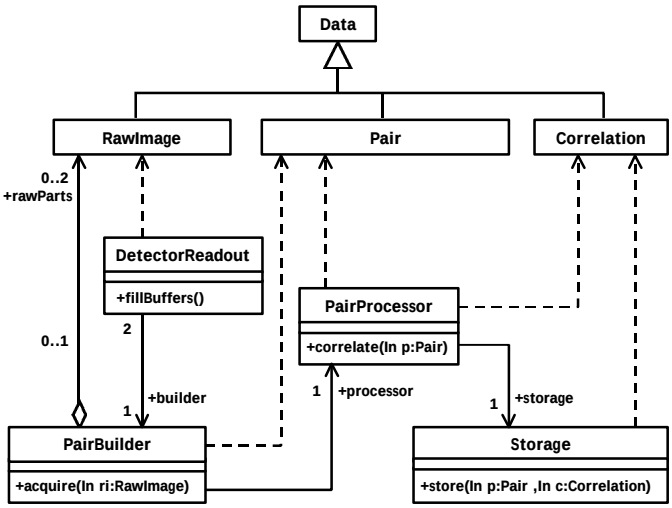

Figure 2: Class Diagram of SAMSA System.

1) Functional Diagrams

Fig.1 and Fig.2 respectively show the sequence diagram that represents the typical call sequence between objects of the system and the class diagram that defines the static architecture of the system. Each detector is coupled with a DetectorReadout object that produces digitized data (images) and sends them (through an asynchronous *acquire(RawImage)* call) to a PairBuilder object. PairBuilder merges each pair of images into one Pair data object and then sends the result to a PairProcessor object through a *correlate(Pair)* call for correlation computation. The Pair object together with the computation result Correlation are then sent to Storage through a *store()* call. These two diagrams are clearly functional specification diagrams, as they define the objects we need and how they interact but do not specify on what hardware infrastructure they are deployed. We could go further and explicitly write all the code attached to the specified classes (for instance, the full code of the PairProcessor.correlate() method). Then, for that single set of functional specifications, the designer imagines two successive different deployment schemes expressed in Fig.3 and Fig.4.

2) Deployment Diagrams

Fig.3 features a deployment where data are produced by "readout," a software component simulating the telescope readout device; the data are then sent through a network

connection to the "processor object" component in which objects PairBuilder, PairProcessor and Storage are implemented. A number of implementation details are stated in this diagram, namely that 1) the PairBuilder, PairProcessor and Storage objects are implemented by the same component on a the same PC-Linux node and consequently run in the same address space whereas the DetectorReadout object runs in another one on another machine; 2) the two nodes running the "processor" and "readout" components are linked by a bi-directional communication package called "myCORBA." The framework can therefore automatically deduce that all method calls between PairBuilder, PairProcessor and Storage objects take place as classical function calls (i.e. through normal post-compilation link), whereas method calls between DetectorReadout and other objects have to go through proxy objects as defined by the "myCORBA" package.

Fig.4 represents a more realistic deployment that is closer to the final system. As expected on most TDAQ systems, the readout of the detector is carried out by a specific firmware, here implemented on an FPGA. The data merging takes place in the "builder" component on an embedded processor running a RTOS (Real-Time Operating System), whereas the correlation computing and the storage are carried out on a PC farm running Linux. Here, classical method calls are only between PairProcessor and Storage objects, the other ones having to go either through a CORBA package or a protocol based on interrupts and shared memory. Moreover, PairProcessor and Storage objects are distributed over a PC farm, which calls for a processor farm management system.

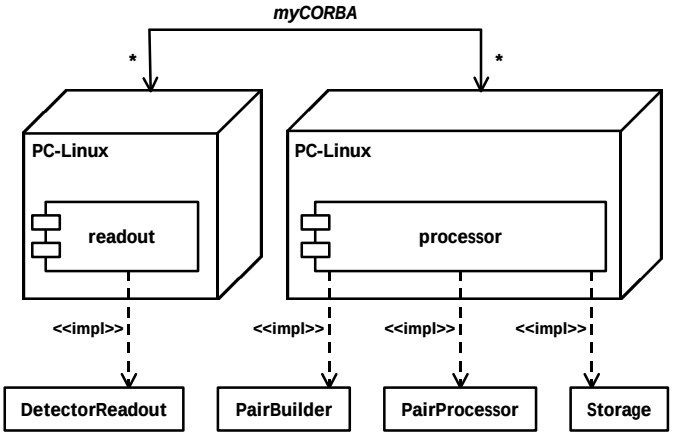

Figure 3: First Deployment of SAMSA System.

3) Needed UML Extensions

Our specific interpretation of node associations as pointers to communication packages is a first implicit UML extension. Other than considering the association name as a communication package name, we also agree upon interpreting the navigability of node associations (arrows at one or both ends of the association line) as directionalities that the communication package can supported. In Fig.4, for instance, the communication link between the ALTERA20K node and the PowerPC node is mono-directional. In other words, only a sender in the "readout" component and a receiver in the "builder" component have to be implemented, as opposed to the CORBA package that supports communications in both directions between nodes PowerPC and PC-Linux. We could render our interpretation of node associations more explicit by creating a new stereotype for them (such as << **comm** >>) in order to avoid any confusion with other interpretations, but we must also try to limit UML extensions to the most needed features and refrain from terminological inflation.

Apart from that specific interpretation of node associations, we have introduced a few specialized stereotypes to be able to specify unambiguously some features in our deployment specifications.

The << **impl** >> stereotype over "use" dependency links is an extension of the UML that we need to specify object implementations in the form of components running on a specific node.

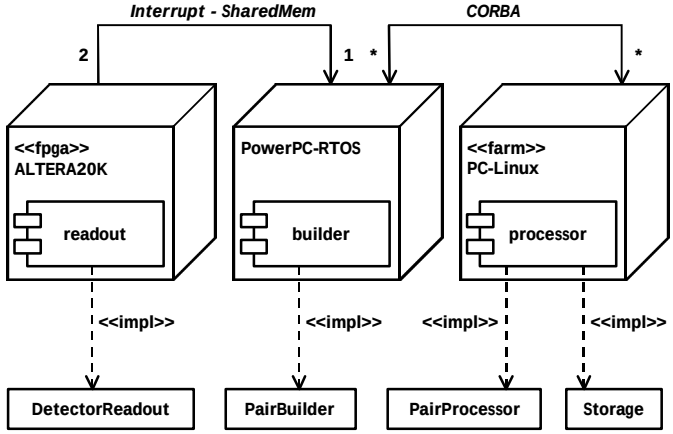

Figure 4: Second Deployment of SAMSA System.

The purpose of the << **fpga** >> stereotype over a node name is to introduce the notion of firmware in system deployment; this allows us to specify deployments featuring objects implemented in hardware (or rather firmware). Any code generator included in the framework would then know which source code type is related to which objects. For instance, in Fig. 4, the "readout" component runs on an FPGA node: a code generator would then use the associated VHDL files to generate the code attached to the DetectorReadout class. In a more distant future, once the UML action language [15] is sufficiently specified by the OMG and developed by the software industry, it would be natural for any code generator to be able to translate action language statements into the right source language (VHDL, C++, Java, etc.) for each object, according to its placement in the deployment diagram.

The << **farm** >> stereotype over a node is of a more subtle nature: it means that although an object such as PairProcessor is seen as *one* object in the system (see in Fig.2 the '1' cardinality in the association between classes PairBuilder and

PairProcessor), it is implemented in Fig.4 on *many* nodes in parallel for the sake of performance and/or failure tolerance.

IV. Automatic Control of Distribution Patterns

Although the specification of design architecture using a formal language such as UML diagrams is in itself useful for productivity and software quality [13], our goal is to be able to achieve more than that. Indeed, apart from presenting a constraining environment to enforce rigor in design, the TDAQ design framework should also, through the cross-analysis of the specification diagrams, 1) check the consistency of our design according to domain-specific criteria and 2) execute automatically model transformations that correspond to the application of recurrent patterns.

A. Consistency Check

Numerous consistency checks can be run on a UML design tool, and most of the industry's CASE tools such as Rational Rose [12] or Objecteering [13] include a number of them, such as namespace and scope coherence. We might include some more that are directly attached to our development model. For instance, the analysis of functional diagrams points to objects that need to communicate with each other (such as PairBuilder and PairProcessor in Fig.1 and Fig.2). Therefore, on deployment diagrams, we can check if any two nodes that run implementations of two such objects are indeed associated. Other consistency checks are possible, especially during automatic model transformation, and they include checking the existence and conformity of communication packages.

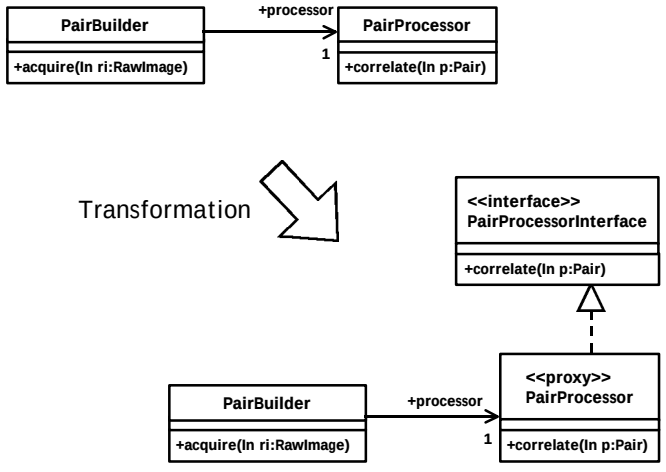

Figure 5: SAMSA Model Transformation on PowerPC-RTOS Side.

B. Automatic Model Transformation

When a remote communication between two objects is detected, the software organization must be modified accordingly. Let us consider again the SAMSA example in its Fig.4 deployment. From Fig.4, the tool can readily deduce the existence of two address spaces, one on an RTOS-running PowerPC, and the other on a Linux PC. It can therefore create one directory associated to each one of them, corresponding to one executable binary for each.

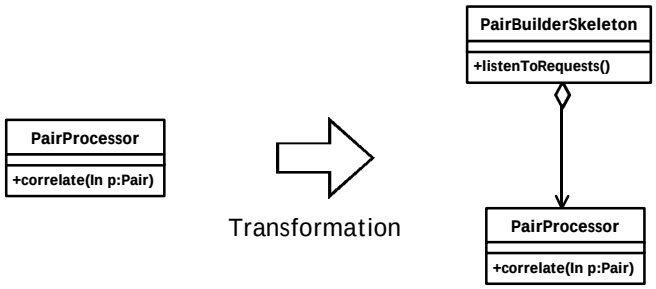

Figure 6: SAMSA Model Transformation on PC-Linux Side.

1) Proxy Generation

Let us focus on PairBuilder and PairProcessor objects. In the PowerPC-RTOS directory, the PairBuilder code is generated using directly the code developed in the functional specification. The same is done in the PC-Linux directory for the PairProcessor code. However, special code must be inserted in the PowerPC-RTOS directory so that all PairBuilder calls to PairProcessor are transparently compiled and run without modification of the PairBuilder code. As expressed in Fig.5, that special code consists in a "proxy" PairProcessor, in the sense defined by ORB architectures (such as CORBA) [5]: it is a class that has the same name as the original class (PairProcessor), the same interface, but not the same implementation code, as the implementation of all methods consists only in the marshalling of parameters, their sending to the real object through the communication package, waiting for the real execution to complete, and finally returning the return value to the calling object.

2) Skeleton Generation

Symmetrically, in the PC-Linux directory, a "skeleton" code (in the CORBA sense, see [5]) is added to the package, that is an object that listens to the communication link for executions requests and translates those requests in actual method calls on PairProcessor. It should be noted that the communication package does not only provide the framework with precise proxy/skeleton production rules, it also has to implement an initialization procedure that correctly instantiates them (a more thorough examination of instantiation question is beyond the scope of this paper).

3) Farm Management

The functional diagrams state that the PairProcessor object is supposed to be logically one single object. However, the << **farm** >> stereotype in Fig.4 indicates that it is implemented as many identical components running on parallel nodes. Consequently, before being able to send the "correlate" request, a PairBuilder object must first determine which component will be the receiver. Inserting a new

management object between PairBuilder and the PairProcessor proxy naturally solves the problem (Fig.7). In other words, the PaiBuilder object will see the farm management object as a genuine PairProcessor (same interface) but its *correlate()* requests will be routed to the right proxy according to the farm management policy implemented inside the manager object. In other words, the model transformation fools the caller (PairBuilder object) by presenting it the interface it expects and hiding the parallelism management issue from it, thus preserving the separation between functional code and farm management code.

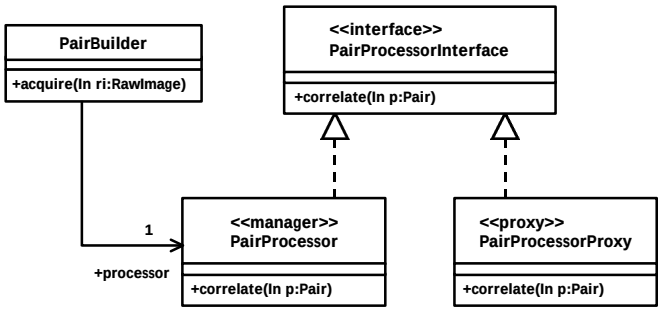

Figure 7: SAMSA Model Transformation on PowerPC-RTOS Side With Farm Management.

V. WHICH SOFTWARE TECHNOLOGY TO USE?

The general cost-reduction pressure that is always present in HEP projects calls for the use of COTS products and industry standards. Indeed COTS theoretically allows for less development and maintenance effort since these activities can in principle be partially delegated through the use of industrial ready-made products. But the same concerns arise when COTS becomes synonym of dependency towards a specific vendor. Indeed, development and maintenances efforts are liable to rise abruptly if the vendor we depend on ceases to support a product or simply ceases to exist. Therefore, we consider that advantages of COTS should be evaluated against dependency problems.

In any case (COTS or not), dependency concerns are less constraining if software sources are accessible (and understandable). The necessity of having access to software sources also stems from the need for performance optimizations, and porting constraints. If the software architecture is well designed and modular enough, performance bottlenecks can be diagnosed relatively easily and more effort can be put in the optimization of small modules. At the same time, good architectural design allows for easy porting of the software over evolving platforms. A clear example of successful software architecture both in terms of porting and optimization capabilities is the open-source CORBA middleware "TAO" developed by the University of Washington [5],[16],[17]. We intend to use this middleware in our own framework because 1) TAO's open-source model together with an abundant documentation allow us to avoid re-developing important amounts of software without being dependent on the goodwill of a vendor, and 2) TAO implements a successful industry standard (i.e. CORBA) that does not depend on any proprietary choice. Moreover, the design of TAO has been strongly constrained to support real-time distributed systems, as opposed to most COTS ORBs, which are known to behave poorly in real-time environments [5][11].

As for the implementation of the design patterns that will be the core of our framework, we need a pattern language based on the UML. As stated in section IV, real productivity gains can be obtained if the framework is implemented on a CASE tool with automatic code-generation capabilities. "Objecteering", by Softeam, is the tool we have chosen for this implementation and is the only vendor-dependant product that we intend to use. The main feature that has caught our attention is a "java-like" pattern language (the J language) that allows the users to freely implement their own design patterns by working directly at the metamodel level, i.e. *before* the code generation level (of course code-generation is, *a fortiori*, also modifiable). In other terms, the tool supports the UML extension mechanisms, and the automatic model transformations that translate TDAQ design pattern can be readily programmed at the metamodel level. As witnessed by the fact that "Rational Rose" [12], the most popular tool in the industry, does not support metamodel programming, CASE tools with metamodel programming capabilities are not quite rare. Apart from Objecteering, a research project called UMLAUT [14], developed by INRIA/IRISA at Rennes in France features analog capabilities although not at the same industrial maturity yet. Since it is an open source research project, switching to UMLAUT is certainly an option that is worth studying. Since both tools support "XMI" the XML extension devoted to the exchange of UML models, the switching should not be too painful.

Eventually, we intend to map our TDAQ design framework to an official UML profile, that is, a domain-specific extension of the UML adapted to the design of HEP-like TDAQ systems.

VI. CONCLUSION AND FUTURE RESEARCH

A. Simultaneous Implementations

A TDAQ design framework that follows the ideas presented in this paper would above all constitute a conceptual environment enforcing an iterative design and development process. Apart from guiding the designer along a progressive path from simple seminal setups to the fully integrated real TDAQ system, it allows the simultaneous maintenance of different implementations of the system. The best example of such a feature stems from the need for physics analysis to maintain a functional simulation of the TDAQ system. Indeed HEP experiments almost always need to determine the precise effect of the TDAQ system on data

quality and for that, physicists need to cross-examine the functional behavior of the system with Monte Carlo simulations of the detector. By itself, such a need calls for the coexistence of at least two deployments of the same system: the real system and a purely functional deployment on a single machine. It may also be necessary to maintain intermediary deployments corresponding to a subset of the TDAQ system for post-production debugging and maintenance.

B. More Automated Model Transformations

Model transformations other than class diagram modifications will be also necessary, especially the modification and/or creation of state diagrams included in special objects such as skeletons or farm managers. Indeed such recurrent problems are solved by patterns that are not always restricted to the static class structure of the system and often involve dynamic specifications too, and we intend to go further in this direction. Also, the notion of a distributed object (like the PairProcessor object) over a farm of processing nodes deserves to be extended to other useful concepts such as the distributed state-machine over all the nodes of the system. These ideas will be investigated further and will fomr the subject of our next paper.

VII. REFERENCES

[1] M. Jacob, "From Basic Research, Its Primordial Goal, to Technological Transfers and Industrial Spin-Offs," *Beaune 97 Proceedings of the Xth IEEE Real Time Conference*, pp. xii-xvii, September 1997.

[2] J. Knobloch, "ATLAS Computing," *Proceedings of CHEP97 Conference in Berlin*, April 1997, http://www.ifh.de/CHEP97/papers/500.ps

[3] J.-P. Briot, R. Guerraoui, "Objets pour la programmation parallèle et répartie: intérêt, évolutions et tendances," *Technique et science informatique*, Vol. 15 – n°6, 1996.

[4] M. D. Lubars, N. Iscoe, "Frameworks Versus Libraries: A Dichotomy of Reuse Strategies," *Proceedings of WISR'93, 6th Annual Workshop on Software Reuse*, Owego New York, November 2-4 1993.

[5] D. C. Schmidt, D. L. Levine, C. Cleeland, "Architectures and Patterns for Developing High-Performance, Real-time ORB Endsystems," *Advances in Computers*, Academic Press, 1999.

[6] H. A. Schmid, "Systematic Framework Design," *Communications of the ACM*, Vol. 40 n° 10, October 1997.

[7] J. Bosch, "Specifying Frameworks and Design Patterns as Architectural Fragments," *Proceedings of TOOLS ASIA '98*, 1998.

[8] D. Roberts, R. Johnson, "Evolving Frameworks – A Pattern Language for Developing OO Frameworks," *Pattern Languages of Program Design 3*, edited by R. Martin et al., Addison-Wesley, 1998.

[9] A. Airapetian et al. (ATLAS Collaboration), "ATLAS Computing Technical Proposal," *CERN-LHCC*, 96-43, December 1996.

[10] S. Anvar, H. Le Provost, F. Louis, "The ANTARES Offshore Data Acquisition: A Highly Distributed, Embedded and COTS-based System," *Proceedings of the 2000 IEEE Nuclear Science Symposium and Medical Imaging Conference*, October 2000.

[11] F. Kuhns, D. C. Schmidt, D. L. Levine, "The Performance of a Real-Time I/O Subsystem for QoS-Enabled ORB Middleware," *Proceedings of the 1999 International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, 1999.

[12] T. Quatrani, "Visual Modeling with Rational Rose 2000 and UML," Addison-Wesley, 1999.

[13] Philippe Desfray, *UML Profiles and the J language Total control over application development using UML*, White paper, http://www.softeam.fr/us

[14] W.M. Ho, J-M. Jézéquel, A. Le Guennec, F. Pennaneac'h , "UMLAUT: An Extendible UML Transformation Framework," *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, October 1999.

[15] Object Management Group, "Action Semantics for the UML – Request for Proposal," September 1999, http://www.projtech.com/pubs/xuml/rfp.pdf

[16] http://www.cs.wustl.edu/~schmidt/ACE.html

[17] http://www.cs.wustl.edu/~schmidt/TAO.html