The evolution of the distributed Event Reconstruction Control System in BaBar

A. Ceseracciu, SLAC, California, USA
T. Pulliam, Ohio State University, USA
for the BaBar Computing Group

Abstract

The Event Reconstruction Control System of the BaBar experiment was redesigned in 2002, to satisfy the following major requirements: flexibility and scalability.
Because of its very nature, this system is continuously maintained to implement changing policies, typical of a complex, distributed production enviromnent.
In 2003, a major revolution in the BaBar computing model, termed Computing Model 2 (CM2), brought a particularly vast set of new requirements in various respects, many of which had to be discovered during the early production effort, and promptly dealt with. Particularly, the reconstruction pipeline was expanded with the addition of a third stage. The first fast calibration stage was kept running at SLAC, USA, while the two stages doing most of the computation were moved to the 400 CPU reconstruction facility of INFN, Italy.
In this paper, we summarize the extent and nature of the evolution of the Control System, and we demonstrate how the modular, well engineered architecture of the system allowed to efficiently adapt and expand it, while making great reuse of existing code, leaving virtually intact the core layer, and exploiting the "engineering for flexibility" philosophy.

EVOLUTION OF A SOFTWARE SYSTEM

The focus of this paper is the evolution of a software system. The system is the Control System for BaBar Event Reconstruction. One original design requirement of this system directly concerned its evolution: *flexibility*.
We discuss here how designing for flexibility affected the actual evolution of the system. The most effective solutions will be highlighted.

THE BABAR EVENT RECONSTRUCTION AND THE NEED FOR A CONTROL SYSTEM

The Prompt Reconstruction (PR) system is part of the software for the *BaBar* experiment. Its tasks are to generate calibrations, and to process the incoming data from the detector performing a complete reconstruction of physical events producing results ready for physics analysis. These two tasks are done in two separate *passes*: the Prompt Calibration (PC) pass, and then the Event Reconstruction (ER) pass.
The data obtained from the detector Data Acquisition System (DAQ), are stored in files using a format known as "extended tagged container" (XTC file). There is a 1-to-1 correspondence between data runs and XTC files.
The Prompt Calibration pass calculates and stores the new calibrations for each run in the calibration database, described in [1]. These calibrations are then used by the Event Reconstruction pass of the run which produces fully reconstructed events and writes them to the Root I/O event store database. Event Reconstruction is processed by multiple farms of 64 to 100 CPUs.
Each farm processes one run at a time. For each run, an instance of a daemon known as the Logging Manager (LM, see [2]) is started on a dedicated server. It distributes the events read sequentially from an XTC file to multiple reconstruction processes running on the farm machines (nodes). The Prompt Calibration farm must process runs in sequential order, as the calibration uses information from the previous runs, while the Event Reconstruction farm is only constrained to process runs that have been calibrated.
A detailed discussion about the BaBar Prompt Reconstruction model is given in [3].
The Event Reconstruction Control System automates the operations to make it possible to process long sequences of data runs with minimum human intervention. It has been presented in [4].

FLEXIBILITY

Defining Flexibility

Flexibility is defined in [5] as the quality of being adaptable. This is the same quality that allows living entities to increase their fitness to their environment, which is, to evolve.
We propose here a metric for evaluating flexibility in software. The idea is to consider the overhead in run-time performance or development time introduced by increase in complexity.
We may then express flexibility F as a function of the average time τ needed to add a new feature or to fix a bug, and the complexity C: $F = \frac{d\tau}{dC}$. A basic metric for complexity C in software is the number of lines of code (LOC).
An *ideal* flexible system is one in which the average effort required for adding a new feature or fixing a bug is constant, regardless of the complexity. That is: $F = 1$.
Common programming experience tells that real systems tend to be far from ideal, or, $F >> 1$. Every programmer knows that it is more difficult to debug a 1,000 lines long program than a 10 lines one.

We will use this metric to qualitatively evaluate the flexibility of our Control System. We don't have enough statistics to provide a number for F.

The requirement of flexibility

Flexibility is clearly a valuable quality for any software system. It must be paid for, though, in terms of design and development effort. Hence, when specifying requirements, it should be stated the extent and domanins in which flexibility is needed.
The case of our Control System needs flexibility for a number of reasons:

- **complexity**: it is a distributed system. Such a system is inherently complex, and flexibility is a most effective means to control complexity. It restrains local development and changes from affecting, and potentially harming, different parts of the system.

- **evolution in time**: it was known in advance, from experience with the previous system, that pressure for changes over time would be constant and demanding. Rapidity and effectiveness of adaptation to new requirements was deemed of great importance at the design level. In fact, during two years of lifetime, the codebase of the system has been steadily increasing by adding new features.

- **evolution in space**: being able to run on different hardware and software infrastructures, with different constraints, software installations, and platforms, was a requirement to make distributed data production possible. It was also deemed important to support distributed development, isolating components specific to a local infrastructure.

- **human interface**: changes in the setup and configuration of a running installation of the system must be simple and safe enough to be performed by relatively untrained "on shift" operators, and at the same time powerful enough to grant a wide range of tuning.

DESIGNING FOR FLEXIBILITY

Flexibility is a concern at any design level. At the level of bare code, we observe that good code tends to be flexible, or, flexibility is one of the metrics against which quality of code is commonly evaluated. At higher design level, though, some patterns and considerations are particularly relevant: modularity, protocols and interfaces, configuration, abstractions, anticipation. We will consider all of them here.

Modularity

A modular system is one in which the composing pieces are standardized and can be assembled in ways not foreseen by the designer. In a software context, this can be translated into coding components rather than applications, and ways to assemble them into a full system.
The main design concept to support modularity in the Control System is the Light Processing Framework (LPF). It defines a bare operating environment and an abstract interface for modules. The modules are then the standardized components as in the previous definition. Every agent process in the Control System is started by a single procedure that instantiates the LPF and loads a custom configuration on it.
The LPF is "Light" meaning that it provides very minimal services, namely: a message passing service between loaded modules, cooperative multitasking, static loading of core modules and startup configuration.
Other services are taken care by core modules: inter-LPF communication, transparent proxy service based on a Naming Service, controlled execution of external commands, and many others.

Protocols and Interfaces

A modular system helps limiting the complexity of individual components by limiting their domain of interaction with the rest of the world to their interface. This is not sufficient to say that it guarantees a system to be flexible: the complexity of the interaction between components can easily become the next limitation to the overall flexibility of the system.
A clean interface design is important to control interaction complexity, but not sufficient. Modeling interaction between components as a set of multi-party protocols helps in understanding the dynamic relationship between them, and in maintaining a flexible system.

Configuration

A modular system is made of components and a technology to assemble them together. This technology is the configuration system. The power and quality of the configuration system plays a major role in the success of a modular design.
In a distributed system, the configuration is responsible not only of assembling components into entities (processes), but also entities into a whole distributed system.
In the Control System a computing farm is characterized by a unique set of configuration files. These files contain information of different levels:

- configuration of individual modules

- definition of entities (LPFs) as sets of configured modules

- binding of entities to host names

- definition of services as sets of bound entities

- hierarchical definition of higher level services as a set of other services

```
state: CreateNodeEnv {
    isA: OprRPStateCreateNodeEnv
    onTransition: OK      goTo: DistributeNodeEnv
    onTransition: Failed  goTo: FatalError
    onEntry: OnEntry
    timeout: 300
}
```

Figure 1: The definition of a single state in the FSM description.

- definition of a system as a set of services

The configuration information includes the topology of the farm, and it is used for the activation of the system. A full system, spanning tens of nodes, can be activated (booted) by issuing a single command. The hierarchical description of the topology of the farm is used to determine the startup sequence, where each service is responsible for spawning its sub-services.
The XML language is a natural choice for describing a highly hierarchical structure such as our configuration. Additional benefits of using XML are the availability of macros, and combining different source files with consistent handling of name spaces.
Common sense rules also apply, with special relevance: no information is to be hardcoded, and the effect of any configuration key must have well defined boundaries, e.g. a key relevant for a given module should not be seen by any other module.
There is in general a tradeoff to be considered between power and complexity of the configuration system: a complex configuration structure is also more difficult to maintain. From our experience, we can state that for a production system, putting complexity in the configuration has been a rewarding choice.

Abstractions

Extensive and proper use of abstractions in software design has a direct impact on flexibility: good abstractions allow us to separate implementation from policy, and to express the evolutionary pressure into new components and policies, rather than modifications (hacking) on the existing code and infrastructure.
The main high level abstraction in the Control System is the Finite State Machine (FSM) processing model. The FSM model allows to split the high level task of processing a unit of data (a *run*) into a set of steps, or states, approximately organized in a sequential manner. Transition to the following state is conditioned to the outcome reported by the previous. The typical semantics here is success/failure.
The definition of the FSM is coded in a custom language. It is a simple stateless language, in which FSM states are bound to physical code modules, and the transitions between states are defined. Figure 1 displays the actual definition of a single state. The FSM also serves as a user interface: it provides a transparent view on the current status of a processing farm, and is used to interact with it. A typical pattern of interaction is: finding out the farm is stuck in one particular state, which gives a hint of what the problem could be; fixing the problem; resume operation of the farm by forcing the FSM to re-execute the same state.

Anticipation

Anticipation is to foresee future paths of development. This is important to understand where in the system flexibility is most needed.
Studying this for our Control System was somewhat simpler, as it was designed to replace a previously existing system. Areas deserving attention were identified simply for being recurrent problematic areas in the old system.

EVOLUTION

We use the word “evolution” as a synonym for “maintenance”. That is to underline that any system under maintenance actually changes during time, and that is why a planned and controlled evolution has a great importance in maintaining or increasing the original qualities of a software system.

Maintenance of a software system

Maintenance, like evolution, can happen in two regimes: ordinary, or regular; and extraordinary, or major.
Ordinary maintenance is concerned with incrementally adapting a system to variations in the environment. This includes adding small features and fixing bugs. A system hopefully spends most of its time in this regime. Typical examples of ordinary maintenance in the Control System are: implementing policy changes by modifying the FSM description; adding new user commands to automate most common commands patterns; supporting different I/O handlers.
Extraordinary maintenance involves adapting a system to substantially different tasks or requirements. As a result of this transition, the shape of the system itself changes, a great amount of code is added, and much code is also removed or, at least, effectively disabled. It is often the case that the “old” system has to be maintained in operation, alongside the “new” one, often for the sake of having a tested backup system.
Often, a software system loses many of the qualities of the original design during a phase of major maintenance, most notably, encapsulation and isolation between components. Two causes for this degradation can be pointed out: insufficient flexibility in the original design, and inadequate care in the new development to respect the original design guidelines. Often, the latter conditions is due to not realizing that a given maintenance task is in fact “extraordinary”.

Extraordinary maintenance: a case

We will discuss now a case of extraordinary maintenance as an example: the upgrade of the Control System due to the advent of BaBar Computing Model 2 (CM2) [6].
From the Control System point of view, this upgrade can be summarized as: moving from a centrally managed event-store database to production and management of plain files. So, a consistent amount of code dealing with database management became obsolete, while much new code had to be added for ensuring integrity of data files and performance of the event store. Moreover, a complex and time consuming post processing stage had to be introduced to merge the partial event data files produced by the individual processing nodes into the final packaged data files to be shipped to the central event store.
Under the evolutionary pressure brought by these new requirements, the main lines of intervention on the Control System codebase has been the forking of the FSM definition to implement the new processing model while keeping the old one untouched; and coding the logic for the individual new operations to be performed into new states. For the post processing stage, an independent system was designed, using the same core infrastructure but separate application code. A filesystem based interface to the main processing system was designed, in order to ensure data consistency between the two stages and allow the number of post processing systems to scale depending on the needs. The upgrade was quickly implemented at a basic production level and was ready in time for production deadlines. Later development towards improved error checking and data integrity insurance was conducted and deployed while the system was in production mode, smoothly and with very little negative impact on the production effort. At the same time, the old processing model was always available without needing to freeze older versions of the code base.

Codebase evolution

Figure 2 reflects the different development phases over the lifetime of the Control System: during Core Development there is development only on the core package, at high rate. In Application Development the core development slows down, while the application code grows extremely fast. Three months later the system starts to be used in production: application development slows down, and most of the new code addresses issues emerging from the real-world test. Both packages see the same amount of activity. Few months later the main development effort is considered done, and the system is Stable and works in a maintenance only regime. There is very little code contributed in this period, the curves are almost flat. This also corresponds to developers switching to other projects in the same period. Then, a major change in the experiment's Computing Model (CM2) requires a major update to the Control System. It is remarkable that very little is done on the Core package, while the Application package grows fast. This shows how much the core code was reused, and

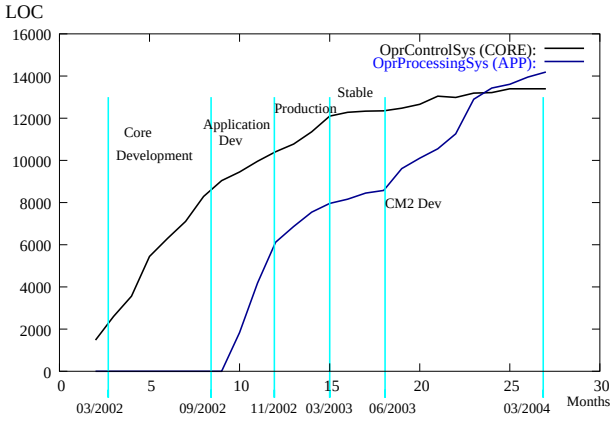

Figure 2: The number of lines of code (LOC) in the two packages (Core and Application) of the code repository, as a basic metric for complexity. The plot reflects the different development phases.

how the flexibility of the core allowed a rapid adaptation of the system to a changed environment. Also, it is interesting to note that the allocated development resources for the project steadily decreased from an initial 3 FTE to a current 0.5 FTE.

CONCLUSIONS

This paper presents the Control System as a self-evolving system under the pressure of environmental changes. This artifice was to illustrate the importance of the requirement of flexibility for a production system. In reality, any stage of the system's maintenance requires thoughtful decisions, to maintain the code quality and the consistency with the original design. High level design concepts and frameworks offer an extremely valuable guidance to those decisions during the lifetime of the system, but, they can not make the system immune to the degradating effects of bad design or bad coding taking place during its lifetime. The experience with the BaBar reconstruction Control System displays how careful design and maintenance can grant a valuable quality like flexibility to a software system.

REFERENCES

[1] I. Gaponenko, *CDB - Distributed Conditions Database of BaBar Experiment*, CHEP2004, Interlaken (Switzerland), 29 Sep 2004

[2] S. Dasu, J. Bartelt, S. Bonneaud, T. Glanzman, T. Pavel, R. White, *Event Logging and Distribution for BaBar Online System*, CHEP98, Chicago (USA), 31 Aug-09 Sep 1998.

[3] A. Ceseracciu et al. *Distributed Offline Data Reconstruction in BaBar*, CHEP2003, San Diego (USA), 24-28 Mar 2003.

[4] A. Ceseracciu, *The new BaBar Data Reconstruction Control System*, CHEP2003, San Diego (USA), 24-28 Mar 2003.

[5] Wordnet - http://www.cogsci.princeton.edu/ wn/

[6] P. Elmer, *BaBar computing - From collisions to physics results*, CHEP2004, Interlaken (Switzerland), 27 Mar 2004.