World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:4, 2007

Toward an Architecture of a Component-Based System Supporting Separation of Non-Functional Concerns

Jerzy Nogiec and Kelley Trombly-Freytag and Shangping Ren

***Abstract*—**The promises of component-based technology can only be fully realized when the system contains in its design a necessary level of separation of concerns. The authors propose to focus on the concerns that emerge throughout the lifecycle of the system and use them as an architectural foundation for the design of a component-based framework. The proposed model comprises a set of superimposed views of the system describing its functional and non-functional concerns. This approach is illustrated by the design of a specific framework for data analysis and data acquisition and supplemented with experiences from using the systems developed with this framework at the Fermi National Accelerator Laboratory.

***Keywords*—**Distributed system, component-based technology, separation of concerns, software development, supervisory and control,QoS

I. Introduction

Component-based technology have brought a promise of reducing software development efforts and focusing instead on selecting, evaluating, and integrating components [2], [5]. The component-oriented approach to developing software can lead to more reusable, extensible and adaptable software. However, achieving the long-term flexibility and maintainability is conditioned upon providing the necessary level of separation of concerns in the design of the system.

Separation of concerns has been recognized as one of the guiding principles in software engineering decades ago [8]. Decomposition is typically done along one dominant concern, with systems consequently suffering from the "tyranny of the dominant decomposition" [12]. In reality, no single decomposition seems to be always the best, and different concerns are important at different stages in the system's life cycle [11]. As a result, different aspects may play different roles at different times. The major concern used in system decomposition is a functional one that defines required core functionalities of the system and consequently its core architecture. Other, non-functional concerns, although not directly implementing core functionality, could play a significant role in fulfilling all requirements defined for the system. Regardless of the focus of a concern, change in one should not affect other concerns, especially the core functional concerns.

Since separation of concerns is necessary for successful complex component-based systems, one should seek architectural solutions within component-based systems to support it. Separation of concerns may happen along various dimensions [12], each of which may be relevant at different stages of software lifecycle. We focus on the integration and maintenance phases of the lifecycle and propose separation of concerns that allow solutions to be independently built via composition in: exception handling, self-monitoring, reconfiguration, coordination and data processing. The separation of these concerns allows for building and easily altering dynamic behaviors of applications (such as reconfiguration, self-monitoring, self-healing, adaptation solutions etc.), aids in debugging and testing, and provides a foundation for achieving desired quality of service.

The proposed architecture is a multi-plane model based on component interactions. The model is rich enough to allow for independent construction of the features outlined above.

The remainder of this paper is arranged as following. In Section II, we discuss the requirements on components and component-based systems at various stages of a software design and deployment. Section III introduces a component-based model. In the model, components are extended with new characteristics so that their behaviors are observable, tailorable, and composable from both functional and crucial non-functional dimensions. In Section IV, we present an application of the model in the Extensible Measurement Framework (EMS) developed by the authors at Fermi National Accelerator Laboratory. The framework implements the presented model and offers a test bed for experimenting with coordination and separation of data processing from control of data flow and coordination of components. Finally, an overview of related work and conclusions are given in Section V and Section VI, respectively.

II. Requirements over a Software LifeCycle

Component based technology is a big step forward in software industry. It provides the benefits of software reuse, shorter development time, improved reliability, and reduced maintenance and development efforts. In the case of component-based software development, applications are assembled from ready-to-use components. Let us consider various non-functional categories of requirements occurring in the full lifecycle of a software.

A. Maintainability

Development efforts are only part of the software product lifecycle. It has been realized that maintenance efforts surpass development efforts over the complete lifecycle of a software

Jerzy Nogiec and Kelley Trombly-Freytag are with Technical Division, Fermi National Accelerator Laboratory, Batavia, IL 60510
Shangping Ren is with Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616

system [16]. Therefore, to fully utilize the potential of component technology, components must include features that help to improve their maintenance. In other words, components should allow easy identification of the system's or the component's malfunction and localization of the identified malfunctionalities. To achieve these goals, the behaviors of components or the software must to be observable. Typical components do not allow inspection of their internals and limit the developer to only them as black-boxes. In the proposed solution, we have instrumented components with mechanisms that allow us to selectively monitor various aspects of the component's inner behaviors and reason about them independent of the main functional concern of the system. This aids in both integration testing and runtime behavior monitoring.

B. Testability

Elaine Weyuker [1] argues that components should be tested not only when they are developed (unit testing), but also in their application context to ensure that the software performs correctly. Failure to do so may lead to serious malfunctions, as proved, for example, by the spectacular Ariane 5 disaster. Typically, the lack of source code leaves the integrator with only specification-based (black-box) testing techniques. In our opinion, instrumenting components with selectable controllable debug streams pertaining to selected design concerns of the component can significantly improve integration and system testing. To further aid performance testing, a set of quality of service measures can be introduced and components can self-evaluate these measures (see Section II-D).

C. Exception Handling

One of the important issues in the design of systems with high level of availability requirement (e.g., control systems, banking systems) is how to handle exceptions. Solutions range from logging exceptions, error notification techniques (e.g., automated calls, warning messages on displays, audible signals, emails, etc.), to correcting the problem by reprocessing (possibly with the use of different subsystem, or reconfiguration). In our opinion, components, as basic building blocks, should not preclude any of these solutions. Rather, they shall provide a basic standard mechanism to allow these solutions to build upon. Therefore, a mechanism to announce detected exceptional situations as well as allow inspection of the "health of the component" helps to meet the requirement of handling of exceptions. This feature is importance for real-time systems and fault-tolerant systems, as it provides a means for real-time detection and correction of exceptional situations.

D. Supprot for Dynamicity

The required speed of adaptation to changing requirements does not always have to be rapid, but often the speed at which an application can be altered to handle enhancements is critical. Depending on the application, acceptable solutions to changing requirements may range from manual reconfiguration of the system to runtime reconfiguration. The architecture of a component system should allow for constructing various solutions when building dynamic systems, including tailoring of component attributes, selecting different algorithms used by a component, replacing a component and modifying the flow of data between components. The component's design should not preclude any of these solutions. Instead, it should support both offline and online reconfiguration and allow the decisions on reconfiguration to be autonomic or as a result from the user's intervention.

E. Quality of Service

Some applications are required to either optimize or adhere to given limits of various measurable parameters. These non-functional factors are typically referred to as Quality of Service (QoS). The QoS is, as defined by ISO [7], a set of qualities related to the collective behavior of one or more objects. For online and real-time systems it is the time-related characteristics category of QoS (according to OMG's classification [6]) that is of particular interest. By applying compositional reasoning about QoS, one can infer system wide timing properties based on the QoS assured by individual components [5]. Although components can be evaluated and even certified at development time to fulfill some QoS criterion such as maximum processing time, the capability to re-evaluate them in a concrete environment and dynamically react to the changing QoS requirements is of high value. Therefore, the capability for runtime evaluation of QoS measures of components, in addition to using design time QoS information, is essential for some class of applications, such as real-time applications.

III. Model

A model of a system represents the system's attributes that are significant from the point of view of modeling. In our case, the focus of modeling is on components and their interactions when they form a complete working system. In this section, we present an component-based model that facilitates behavior monitoring, feature extension for accommodating new QoS requirements and system reconfiguration.

A. Component Model

There are many definitions available for components [2], [3], but all seem to agree on that a component is a unit of composition and component-based development are the process of integrating components. Furthermore, one can state that a component provides its functionality via its interfaces. In other words, a component representing a source will implement a producer interface and a component representing a sink will implement a consumer interface.

Unlike traditional components that exchange unspecified typed data, in the model presented here, components exchange five distinctive types of events: data, property, control, exception, and debug. Each component can be either or both a producer and consumer of any type of event. Moreover, to simplify the connectivity, components receive and send events on input and output ports. Consequently, depending on the interfaces implemented, a component can be a data source,

data sink, exception handler, property monitor, coordination agent, etc. A typical functional (data processing) component implements a data consumer, data producer, exception producer, debug producer, and control consumer interfaces. Figure 1(a) gives a visual representation of a typical functional component.

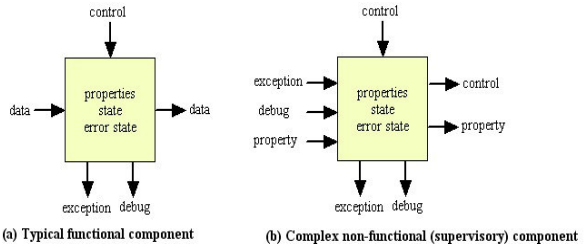

Fig. 1. Visual representation of components

Components implementing the exception consumer interface allow for the building of exception handling solutions based on the exception event streams originated in other components, including filtering exceptions, multi-stage decision systems, logging, reconfiguration etc.. Similarly, components accepting debugging events may be used to construct solutions in observing the behavior of the system, helping debug it, and reason about the state of the system, which can then be used for various coordination decisions.

Following the separation of concerns principle, an application will consist of some supervisory or coordinating components in addition to the component fulfilling the functional aspects of the system. These components will not be engaged in processing data but rather in sending or receiving control or/and property events. Figure 1(b) depicts a supervisory component. Detailed discussion on the supervisory components is given in Section IV-A2. Events exchanged between components are collections of named items, and therefore to some extent self-describing. This, plus a reflection mechanism, allows for implementation of general-purpose supervisory components to persist, examine, present and visualize any event in the system. It also enables functional components to retrieve and process only data items of interest to particular component and easily implement variations of such patterns as chain of command or decorator. It also allows for changing parts of exchanged data for one concern without impacting the implementation and processing of other concerns or features.

Each component has a set of attributes called properties. These properties can be defined as externally modifiable or only readable. In addition, each component also defines a special subset of properties representing its runtime QoS measures. They can be runtime monitored and hence provide a handle for runtime detection of bottlenecks and runtime measurement of the throughput or reaction time of the system.

Martin Griss [4] argues that system complex dramatically increases when components have compile time or runtime customizable property parameters, and especially when properties of several components have to be manipulated in conjunction. To cope with this difficulty, we have developed a mechanism that is based on group property and atomic initialization of components. In this solution, a subset or all of properties are specified as “remote” properties. These ’remote’ properties are atomically initialized from an external source — a component implementing a producer interface and connected to the components that have the ’remote’ properties to be initialized. This mechanism separates all details necessary for tailoring or configuring a component to its new role from the control details that guide the component toward capable of performing a specific algorithm best suited to its current quality of service criteria.

B. System Model

A system is a network of connected components, where each connection or route links together an output port of one component with an input port of another. Though components are capable of accepting any types of events, but they only react to interested data. Connections for a component are separately defined for each type of events and can be visualized as five independent directed graphs, each of them describing a separate concern, and all having the same set of nodes representing components. Not all nodes have to be participate in each concern.

Figure 2 depicts an example of a component system and the communication among components. In the figure, nodes on each vertical line represents a component involved in a concern and arrowed lines shows the role each component plays in different concerns.

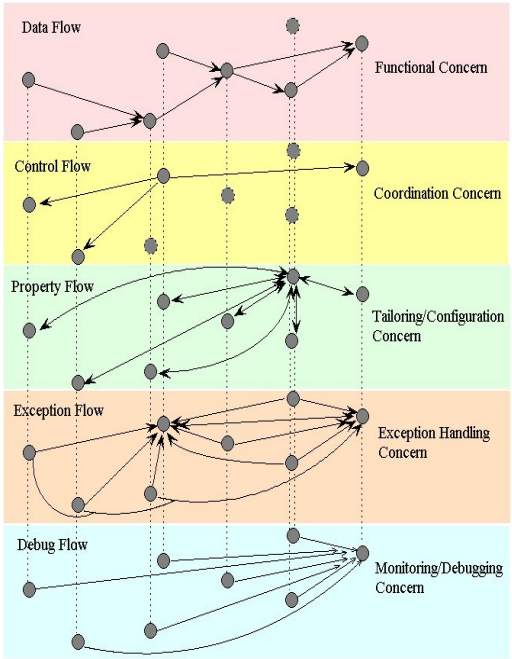

Fig. 2. Visual representation of inter-component communication

This approach separates the processing for different types of events and therefore provides the separation of various concerns, including the main functional concern connected with data processing. Special components allow for translation

of non-data events to data events and therefore allow for the use of the data processing capabilities of components to process other than data events. Modification of the these graphs allows the system to adapt to new requirements, which can be also augmented with blocking data flow in some of the routes in order to divert the data flow.

In next section, we apply the model in the development of Extensible Measurement System (EMS) that is used in Fermi National Accelerator Laboratory for performing superconduct tests.

IV. EMS Framework

The Extensible Measurement System (EMS) framework is developed in response to a need to build a system for research and development environment characterized by high adaptability to changing requirements, rapid development/modification time, and flexibility in building families of related systems. It is built based on the model described above and is targeted at data acquisition, processing and analysis. The following lists the main features that the framework possesses:

- Component technology to allow for reuse
- Architecture that supports extensibility
- Configurable applications
- Tailoring of applications at run-time
- Scripting to rapidly develop or modify applications
- Integrated application monitoring, debugging, and exception handling Highly-configurable components
- Universal components that accept various collections of data items

The complete EMS system consists of components, an execution environment, and various tools supporting development. Figure 3 depicts an application development under the framework. Our focus of this paper is on the components. The following subsection discuss the components, execution environment and tools in detail.

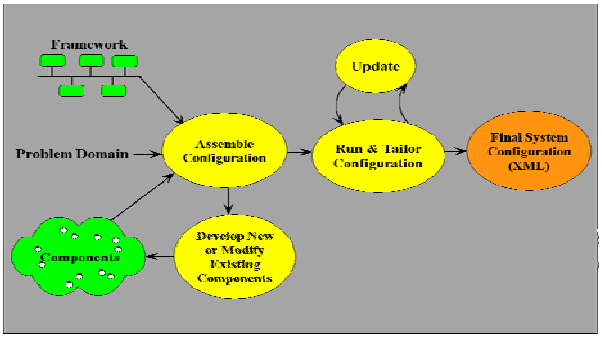

Fig. 3. Application Development under EMS

A. Components

The EMS defines a set of interfaces and abstract implementations that form the basis for implementing the types of components that are need to comprise EMS applications. Components are independently released software module suitable for composition (together with other components) into multiple applications. Components have properties and state. Typical components input, process, and output data. Their behavior depends on their state and property values. Components can be forced to perform certain actions by sending control events to them. They also output debug and exception information.

In particular, in EMS, we have the types of components:

- Core components
  - Application monitoring
  - Scripting
  - Event processing
- Data Presentation
  - Graphing components
  - Display components
- Control and Input
- Data Processing
- Data Input/Output
  - File I/O
  - Database access
  - URL access
- DAQ

EMS provides for independent "wiring" of components for property, data, exception, control, and debug events. Figure 4 depicts an example of an application built through wiring components.

1) Model Implementation and Expansion: EMS expands on the model presented in Section III by defining a hierarchy of interfaces that classify the components into various categories such as data processing component, visualization component, supervisory component, management component, etc. A hierarchy of abstract implementations of these categories has been created in parallel to the interface hierarchy. A set of both universal and specialized components have been developed for particular vertical domains.

Components, depending on their category, have common sets of properties. Two important mandatory properties of each component that can be externally inspected but not manipulated are state and error status. Each component is always in one of its permitted states. The state of the component changes in response to the receipt of a specific control event or as a result of processing an input event. If, during processing, an exceptional situation arises, the component changes its error state to reflect this. This feature allows for easy monitoring or self-monitoring of an application and the implementation of various self-healing strategies, all of which depend on the capability to detect incorrect behaviors of components.

All components have a basic set of state definitions with corresponding control signals. These vary with components' category. They also contain the definition of standard debug categories and functionality to create and communicate debug events of varying granularity. In addition, standardized categories of information severity have been defined, which can be placed into events and directed to monitoring components.

All components in the data processing category are instrumented to evaluate data processing timing, crucial from the point of view of on-line application processing time. This feature enables detection of bottlenecks and estimation of the

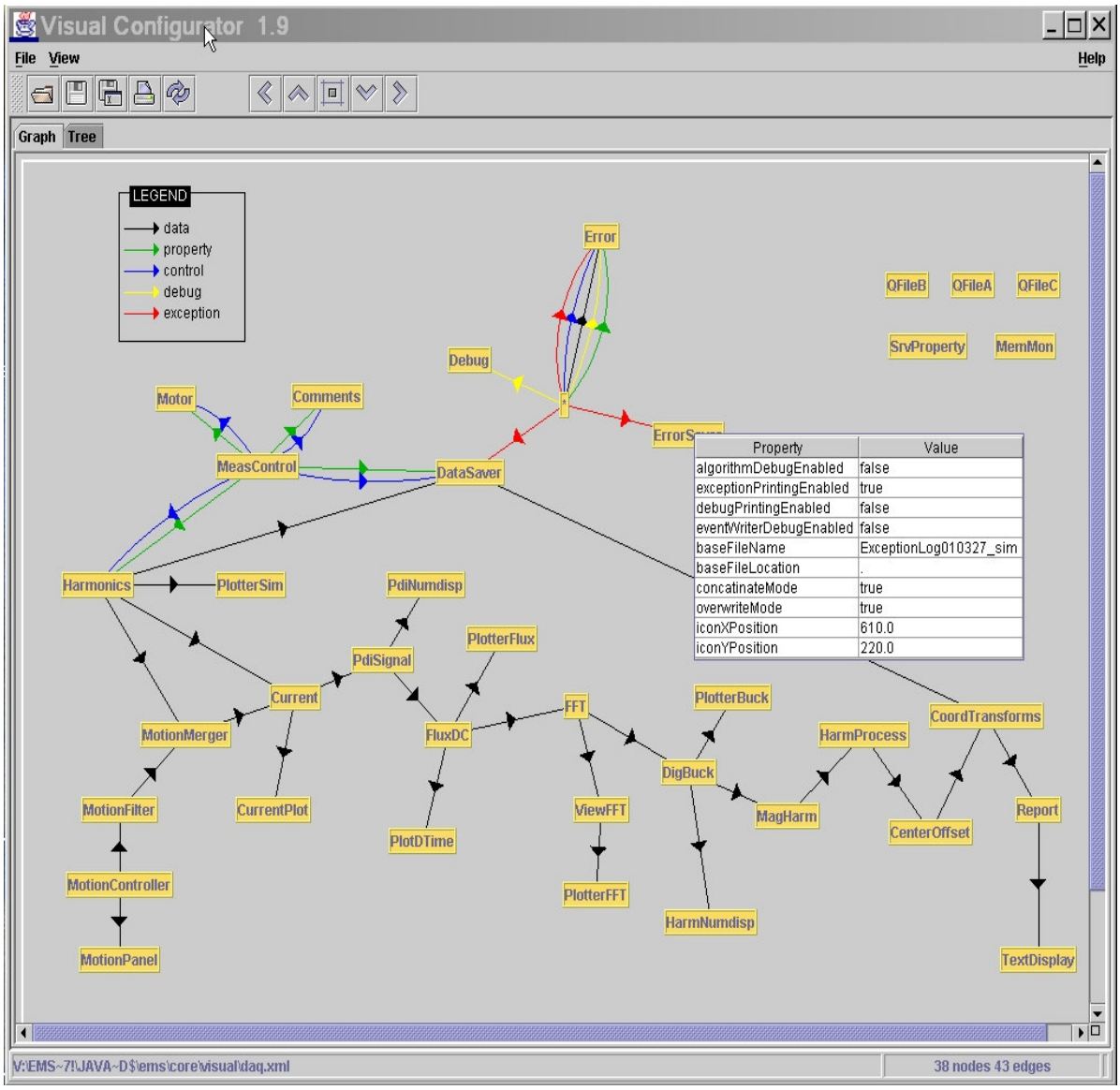

Fig. 4. Wiring components through visual configuration tool

throughput or reaction time of the system, and performing reconfiguration decisions.

2) Types of Components: EMS comes with a variety of components for data visualization, persistence, manipulation of data streams, translation between events types, synchronization between data streams and buffering, as well as monitoring of the components state, manual manipulation of a component attributes, etc.

All components in the framework can be placed either into a supervisory and or a functional category, following the idea of separation of concerns (Figure 5). Supervisory components do not participate directly in processing data but rather are involved in monitoring of other components, reconfiguration of the routing, tailoring of other components, and various coordination functions. A special example of a supervisory component is a multi-language script executor.

Scripting is used as a mechanism of choice for rapid development of new procedures or control sequences in EMS. It is used as a pure control mechanism that does not manipulate data directly, but only triggers and coordinates actions of various components. It can also contain reconfiguration sequences.

EMS supports several mechanisms allowing for dynamic adaptation of an application. One supervisory component that provides manual property changes and examination at runtime is the Property Controller component. This component provides a list of all of the current values of all components

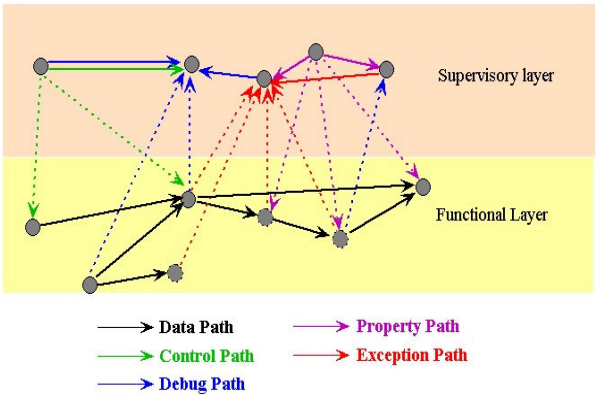

Fig. 5. Separation of concern between supervisory and functional layers of components

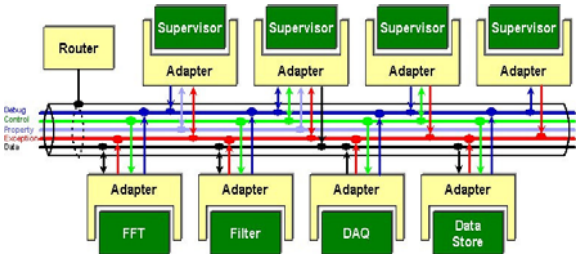

Fig. 6. Communication over a software multi-bus

in the system and allows the user to change these properties while the system is functioning. In addition, it allows for sending of the control signals to a component that has been designed to use. This helps in tailoring and adjustment of a working application, and in integration testing and debugging of a component.

B. Environment

The EMS framework also includes an environment necessary to describe and initiate the system and a runtime support for inter-component communication. The environment supports the communication of various types of events. It also allows for the collaboration of several systems of components distributed between different nodes.

1) Software multi-bus: The major portion of the framework consists of the software bus that supports communication using the various event types as shown in Figure 6. Each of the different type of events (data, properties, etc.) is independently routed between the components in a system. Data events also allow for further specification of their contents by using predefined ports that describes the type of data being moved. The multi-bus supports unicast, multicast and broadcast communication patterns as well as source addressing.

2) Configuration: EMS systems are configured using an XML dialect which also supports the inclusion of sub-definition files and is therefore hierarchical.

Each component is detailed along with its properties. The routing of the various event types between the components is described in the configuration file. The initial state of a component is controlled by defining control signals to be sent to the component. A simple configuration may look as follows:

```
<!DOCTYPE configuration SYSTEM "ems.dtd">
<configuration version="0.1"
               title="Display Test XML">

<!-- Component definitions -->
<component id="Generator"
   class="ems.core.cmpnnts.dsp.DataGnrtr">
  <property name="delay"
            value="10000"/>
</component>
<component id="Display"
   class="ems.core.cmpnnts.SmplDataDsply">
  <property name="title"
            value="Display Component"/>
  <property name="XPosition"
            value="0"/>
  <property name="YPosition"
            value="500"/>
  <property name="wrapLines"
            value="true"/>
  <property name="width"
            value="400"/>
  <property name="height"
            value="250"/>
</component>

<!-- Routing information -->
<route type="Data"
       origin="Generator"
       destination="Display"/>

<!-- Control signals -->
<control signal="init"
         destination="!"/>
<control signal="start"
         destination="!"/>
```

Besides the initial sequence of control signals in the XML configuration, the system basically works similarly to a work-

flow system with components being purely data driven. The necessary control on components states, or data flow pattern, or the activation or deactivation of components is done by supervisory components.

C. Tools

EMS contains a variety of tools designed to aid component developers, integrators, and users of the system. A component introspection tool is used by developers for component verification, allowing verification of the components adherence to architectural requirements. The visual configuration tool provides a graphical view of a set of components, with the routing information displayed as arrows.The component names and different event flow information are displayed in color, and all property values for each component are available for viewing. The graphical display can be manipulated to rearrange the component boxes, and the final combination can be stored with the configuration itself for later use with the tool. Figure 3 is one of the UI interface of the tool. It is a valuable aid for integrators in assembling components into applications.

Another category of tools is focused on preparing and using component documentation. An on-line component configuration and selection tool is available on the web as a standalone application. The documentation categorizes each component and gives specific information on the states the component can achieve and the attributes available to tailor it.

V. Related Work

Separation of concerns has been a topic of interest and research for quite some time, with Parnas' article on decomposing systems [8] being its cornerstone. Recently, it has been an area of intensive research that resulted in, among others, the introduction of Aspect Oriented Programming (AOP) [13]. Inadequate separation of concerns is recognized as one of bigger problems in contemporary software. Harrison and Tarr [11] name it as a major barrier to portable and morphogenic software. The component-based domain does not stay immune to these trends, and the importance of the idea of separation of concerns in combination with component-based technology is an area of active research.

The OMG's Corba Component Model (CCM) [17] introduces a notion of a component container. The container is an intermediary between the component, the Corba ORB and Corba services. A container can be selected from several container types depending on the component's characteristics and it provides non-functional services. The selection cannot be changed at runtime, which limits the dynamic reconfigurability based on non-functional properties.

Duclos and others [18] focus on non-functional aspects in component-based applications and propose a solution that merges a container-based approach to separation of concerns with AOP's approach. They distinguish between aspect designer and aspect users and allow aspect designers to define new aspects and aspect users to apply them to components.

The COMQUAD component model [19] extends the concepts of Enterprise Java-Beans and CCM. Non-functional aspects are "woven" into the running application by the component container. At runtime, the component container is responsible for selecting component implementations to be instantiated based on non-functional requirements including such QoS properties as timeliness, accuracy, precision, and quality of video stream. The focus of the COMQUAD model seems to be on the negotiation of required properties and selection of appropriate implementations.

Sora and others [14] focus on self-customizable systems, that is systems equipped with mechanisms to automatically adapt themselves to a set of user requirements or to their environment. Similar to the EMS framework, they address the adaptability or reconfiguration problems through component composition, but they propose to hierarchically decomposed component systems, which differs from the approaches based on separation of concerns as in EMS.

Nunn and Deugo [15] propose a system in which XML code instructions tell a composition engine how to connect various components to produce a running application. The approach of using an XML-driven configuration is also presented in the EMS framework.

VI. Summary and Conclusions

It is our belief that the concerns emerging throughout the lifecycle of the system should be used as an architectural basis for the design of a component-based framework. The proposed model and framework based focus on separation of concerns in the non-functional areas of testing, exception handling, coordination, and reconfiguration. The emphasis of this model is on the observability of the components and their composability. The result is a framework that is suitable for rapid application development and enhancement,is highly adaptable and easily maintainable.

The framework described has been successfully used to build several applications, some of them have been in use for a while and have shown the advantages in adapting and maintaining the applications. In particular, of special value, especially in R&D environments, is the framework's capability to reconfigure and to easily adapt to changing requirements and the capability to selectively debug and monitor components at runtime. Under this framework, we are able to examine, filter, persist and visualize data originated from any output port of any component at any time, without impacting core functionality of the system. This proved to be a very valuable asset in our Technical Division for R&D purposes.

The EMS is still a very much active project, with its future directed toward dynamic reconfiguration and adaptation for on-line data acquisition and measurements.

Acknowledgements

The authors are grateful to all contributors to the EMS project for their efforts in implementing and shaping the framework. Special thanks go to Dana Walbridge, Sergey Kotelnikov, and Gene Desavouret from the Technical Division at Fermi National Accelerator Laboratory.

REFERENCES

[1] Weyuker E.J.: Testing component-based software: A cautionary tale. IEEE Software, vol. 15, no. 5 (1998) 5459
[2] P. Brereton and D. Budgen, Component-based systems: A classification of issues. IEEE Computer, vol. 33, no. 11 (2000) 5462.
[3] Clemens Szyperski, Component software: Beyond object-oriented programming. ACM Press/Addison-Wesley, New York, NY, 1998
[4] Griss M.: Implementing Product-Line Features By Composing Component Aspects. First International Software Product Line Conference, Denver, Colorado (2000)
[5] Sun C.: Empirical Reasoning about Quality of Service of Component-Based Distributed Systems. ACMSE'04, Huntsville, Alabama (2004)
[6] Object Management Group. UML Profile for Modeling Quality of Service and Fau;t Tolerance Characteristics and Mechanisms. Request for Proposal, OMG document ad/02-01-07, Framington, MA (2002)
[7] ITU-T Recommendation X.641 (1994), ISO IEC TR 13236, Information technology Quality of Service Framework.
[8] Parnas D.: On the Criteria to be used in decomposing systems into modules. Communications of the ACM , vol. 15, no. 12 (1972)
[9] Nogiec J., Sim J., Trombly-Freytag K., Walbridge D.: EMS: A Framework for Data Acquisition and Analysis. ACAT2000, Batavia, Illinois (2000)
[10] Nogiec J., Desavouret E., Kotelnikov S., Trombly-Freytag K., and Walbridge D.: Configuring Systems from Components: The EMS Approach. ACAT'03, Tsukuba, Japan (2003)
[11] Harrison W., Ossher H., Tarr P.: Software Engineering Tools and Environments: A Roadmap 2000. Future of Software Engineering, Limerick, Ireland (2000) 263-277
[12] Tarr P., Harrison W., Ossher H., Finkelstein A., Nuseibeh B., Perry D.: Workshop on multi-dimensional Separation of Concerns in Software Engineering. Software Engineering Notes, vol. 26, no. 1 (2001)
[13] Kiczales G. et al.: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, Springer-Verlag LNCS 1241 (1997)
[14] Sora I., Verbaeten P., Berbers Y.: Using Component Composition for Self-customizable Systems, Proceedings - Workshop On Component-Based Software Engineering: Composing Systems from Components at ECBS (Crnkovic, I. and Stafford, J. and Larsson, S., eds.), Lund, Sweden (2002) 23-26
[15] Nunn I., Deugo D.: Automated Assembly of Software Components Based on XML-Coded Instructions, SAC 2002, Madrid, Spain ACM (2002)
[16] Guimaraes T.: Managing application program, maintenance expenditures. Communications of the ACM, vol.26, no. 10 (1983)
[17] Object Management Group. CORBA Components (2001)
[18] Duclos F., Estublier J., Morat P.: Describing and Using Non Functional Aspects in Component Based Applications. AOSD 2002, Enshede, The Netherlands (2002)
[19] Goebel S., Pohl C., Roettger S., Zschaler S.: The COMQUAD Component Model, Enabling Dynamic Selection of Implementations by Weaving Non-Functional Aspects. AOSD 2004, Lancaster, UK (2004)