Practical Software Engineering Support for Component-Based Control Systems.

Francois Bronsard, Gilberto Matos, Dilip Soni
Siemens Corporate Research
755 College rd. East
Princeton, NJ 08540

INTRODUCTION

This position paper presents our current work on development tools for component-based systems. Although some of our efforts are specificly aimed at the field of control systems, we believe that the development tools suggested here can be generalized to other fields in which component-based systems are becoming common.

The field of control systems is experiencing a shift from stand-alone applications to distributed component-based systems. In this new environment, a major challenge for control system developers will be the correct and effective integration of COTS components and custom-made components. Our goal is to produce tools to help developers verify whether their own components make correct and effective use of the environment provided by the COTS components and the underlying infrastructure.

COMPONENT-BASED CONTROL SYSTEMS

Traditionally, small and medium sized control systems are stand-alone programs residing on Programmable Logic Controllers. However, current trends render this simple architecture obsolete. First, at the bottom layer of this architecture, the device layer, the devices controlled by the PLCs are becoming "smarter" and able to take over part of the control logic previously residing on the PLC. Making use of these capabilities would improve control systems. Similarly, in modern systems the top layer of this architecture, the presentation layer, does not have as its sole client a single human-machine interface running on a monitor residing in a centralized control room. Today, many business applications want to access the information present on the PLC and even to exert some high-level control capabilities. Finally, the industry would like more flexibility in setting up and evolving the plants being controlled. One would want to be able to replace a crane in a plant by a newer, improved model without needing a completely new control system. By encapsulating the control of the crane in a separate component, such flexibility becomes possible.

To address these trends, control systems are moving toward a component-based architecture with systems built on top of a standard distributed-enabled execution environment, namely Microsoft DCOM. Thus, we see a future where most components needed for a control system will be provided by various commercial entities. The manufacturers of the devices used in a plant will provide software components to control these devices. Another set of components will be provided by control and optimization specialists. Finally, the plant engineers will add custom

components to address plant-specific issues and to provide the high-level control mechanisms. In this context, the major challenge faced by control system developers is to insure that these components will work together correctly and effectively.

THE INTEGRATION PROBLEM

Our concern is to integrate the COTS components with the custom-made components, and to do so rapidly and reliably. However, the barriers to integration are numerous. Often an in-depth knowledge of the components being used and of the infrastructure supporting the system is necessary to build a truly optimal system. Therefore, the long learning curve for complex components and infrastructure impedes their adoption. To address this problem, we are proposing a tool able to check *usage rules*. We envision that complex components would be accompanied by a collection of usage rules describing how best to use the component. Our tool would then test whether the custom-made components adhere to these rules.

As components and the infrastructure become more complex, the interactions between them also become complex. This leads to a need to document these interactions, and to test whether the client objects of these components use the components correctly. The second tool that we are developing is designed to help test such complex interactions. We envision that components will be accompanied by a description of how the component interacts with its environment. Our tool would generate a run-time mechanism to monitor how the rest of the system interacts with that component and check if this interaction satisfies the specification of the component.

CHECKING USAGE RULES

The COTS components that will form the core of a control system, as well as the underlying infrastructure, form a collection of complex frameworks that plant engineers must master to build a performing control system. However, using complex frameworks is challenging even for experienced developers. Without knowledge of the intricacies of each component and of the infrastructure, many common programming practices either fail to take advantage of the framework, or worse, introduce errors. For widely used frameworks, this leads to the creation of user groups and the publication of FAQs, or even books, dedicated to cataloging "usage tips" for using the framework. For example, to use the Microsoft COM infrastructure more effectively, one might read the recent book "Effective COM: 50 ways to improve your COM and MTS-based applications" by Box, Brown, Ewald, and Sells.

Many of the tips for using a complex component or a framework can be captured by programming guidelines, or usage rules, that can be checked automatically. Guidelines identify those code patterns that are inefficient or incompatible with the framework, and suggest alternative design choices that make better use of it. Automatic guideline checking improves software quality and performance by finding programming errors or inefficiencies. It also reduces the amount of effort needed to develop a system since full mastery of the framework is not required.

We have developed a generic Code Inspector which provides a powerful facility for ensuring that source code adheres to programming guidelines and usage rules. This code inspector has been

used in projects to detect common mistakes and test adherence to project specific guidelines. For example, a version of the code inspector targeted to C/C++ programmers checks guidelines such as "Do not use delete[] to delete a non-array and delete to delete an array." This C/C++ inspector currently checks for 150 guidelines.

In our current project, we are combining the Code Inspector with usage rules derived from Box et al.'s book to develop a small COM Inspector to help developers produce more effective COM programs. For example, one of the guidelines explains how to make use of the facilities of the COM layer to protect clients and servers against communication failure: for a client, it is recommended always to check the return value of remote procedure calls since this value will indicate communication failure; for a server, it is recommended to use the worker pattern since this enables one to use the COM garbage collector to be warned of client or communication failure.

The COM inspector demonstrates the feasibility of building a code inspector targeted to a specific complex framework. We propose to generalize this approach and suggest that components be accompanied by collections of usage rules that could be checked automatically. This would significantly ease the use of complex components.

TESTING

It is often hard to debug a complex application to find out what caused an error, since the application consists of a number of concurrent processes or threads. One way to quickly identify when something goes wrong is to add runtime verification code to the components and detect anomalies as they happen. It is rarely acceptable to use this code in real products due to its overhead, and it is even less practical to require designers to implement and maintain multiple versions of their components. Fortunately, code that verifies expected behaviors can be automatically generated and integrated with the original components, creating temporary objects that log and analyze some aspects of system interaction to detect errors.

The component designer can specify its expectations in the form of a finite state machine whose transitions are enabled by the occurrence of specific events, such as method invocations. This state machine contains specific states that are only reachable as a result of interaction patterns, usually incorrect ones. Logging and monitoring of a component's environment allows the integrators to identify the first time an unexpected event affects a given component, thus guiding them faster toward the original cause of the error. Interaction logging provides information on the coverage of the expected behaviors, and incomplete coverage indicates that a component is not being used to its full potential, suggesting possible efficiency improvements.

We are developing a tool that accepts a UML specification of a component with a state machine description of its environment expectations, and produces monitoring code that can be used as a wrapper for the original component without modifying its functionality. Our tool is specifically directed at COM-based systems, and relies on the interception of interfaces provided by a given server component. The designer of a COTS server component should provide a description of its environment expectations, either in the form of monitoring code or UML state diagrams that can be used to generate the code. The instrumented component monitors the incoming events and

signals to the user if the assumptions are violated. At the end of the execution, the log can show whether all the states and transitions of the expected behavior were exercised by the test drivers.

We are working on broadening the range of events within COM that can be used to monitor expected system behaviors, e.g. by capturing outgoing interfaces and monitoring dynamic object creation. We are also studying the application of this technology to more complex interaction patterns, such as groups of components involved in a use case. We envision that tools like this will be used by COTS components developers to create executable objects that can monitor their environment when necessary, and cooperate with other similar components to verify the correctness of their execution.