Generalized Architecture for Reusable Industrial Automation Software

Carlos O. Morales, Member, IEEE

Abstract—This paper presents a generalized software architecture that addresses the increasingly demanding need for flexibility, robustness and reuse of industrial automation and test software. The proposed architecture has been implemented as a component-based, object-oriented application framework, and new applications implemented with this system have achieved total source code reuse of almost 90% and total software development cost savings of about 70%.

An important feature of this design is a standardized interface specification and software component structure for controlling peripheral devices that are typical to most automated machines. These components are completely independent and can be developed, modified, tested and debugged in isolation from the rest of the system. The system engine provides a multi-threaded platform for the execution of concurrent or sequential tasks.

Note to Practitioners—This work was motivated by the need to optimize resources, cost and quality when implementing new software products for the industrial automation and test domain. Applications are too often created from the ground-up when a new project is started without leveraging solutions that may have been developed for previous systems, notwithstanding the similarities between them. This paper describes a software system that can be reused in multiple applications, thus minimizing development time and cost, while maximizing software quality through extensive reuse. Software reuse is highly desirable not only because of its economical impact, but also for its important influence on the ultimate quality and reliability of the delivered software system. Several successful projects based on this framework have been deployed in the field, ranging from robotic optical device assembly and material handling, to the automated testing of delicate medical instruments.

Index Terms — Frameworks, manufacturing automation software, object oriented methods, software engineering.

I. INTRODUCTION

SOFTWARE reuse is one of the most efficient ways of producing systems that combine robustness, reliability, and high quality, with low cost and short project schedules. A very effective method for reusing software is by developing enterprise frameworks, which are partially finished applications that address a particular domain [4], [13], [16]. Application frameworks reuse not only code, but most importantly, reuse architecture and system design [21], [22], [23].

At PEMSTAR, we have developed a software architecture that generalizes the industrial automation and test domain. This abstraction was evolved and developed by leveraging years of experience in creating diverse applications in this area [14], [18]. This architecture has been implemented as an object-oriented application framework, which in its current state is best described as a gray box framework with pluggable objects, i.e. a mixture of executable modules (black box) and code templates (white box) [14], [16]. This framework has been deployed in the field for several applications, including robotic cells, batch-mode functional testers and interactive product validation and diagnostic systems. These systems integrate a broad range of peripheral devices for industrial automation and test like robots, machine vision, instrumentation equipment, digital and analog data input/output boards, databases and networking.

The software architecture was designed using Information Hiding, Separation of Concerns and Design for Change as guiding principles [10], [27], and it was implemented as an object-oriented enterprise framework made out of software components that are compiled independently [11], [12]. Some of these components can be tested, run and debugged as stand-alone applications. Only at run-time the framework becomes an integrated system of cooperating objects by means of object composition.

The use of this framework as an instance of the architecture described herein enables us to reuse both the high-level design and all the executable code comprising the System Engine. This engine may be viewed as an open system that allows the addition of new industrial automation devices through the interface that we have called ActiveDevice [30]. ActiveDevices are viewed as software assets that are reusable in executable form when plugged into the System Engine.
compose new applications.

The System Engine provides a multi-tasking platform for the execution of concurrent or sequential processes called Application Sequences. These sequences implement custom algorithms, each devoted to a single aspect of a system, for example, controlling a robot or performing a mathematical computation, and they may cooperate and synchronize with one another to accomplish complex tasks. This approach separates concerns and fosters the development of source code that is easy to modify and extend. Changes are thus constrained to specific software components in a loosely coupled, dynamic system. Therefore, the likelihood of propagating side effects to other parts of the system is minimized.

The System Engine includes Graphical User Interface (GUI) components that separate their implementation from the system design. User interfaces are implemented with a standardized main screen that is customizable through external code. Custom GUI components can be added through an external component.

The rest of this paper is organized as follows: Section II introduces the research problem and related work. Section III describes the system architecture. Section IV presents the system’s dynamic operation. Section V shows how this architecture is applied to a sample application. Section VI evaluates the advantages of using the proposed architecture as compared to the traditional approach to software development in the target domain, Section VII presents conclusions, and Section VIII discusses future work.

II. REMARKABLE PROBLEM AND RELATED WORK

Software systems for industrial automation and test equipment are typically developed as part of strongly constrained projects, with short schedules, tight budgets and a high-stress pace. Custom automated manufacturing plays a critical role in bringing new products to the marketplace, and these constraints are merely a reflection of a multitude of forces driving product cost and market opportunity windows.

Under these circumstances, it is always a challenging task to deliver high quality automation equipment that must be designed from scratch, and that has to be done right the first time, and on-time. Software projects, in particular, are infamous for taking as much as three times the originally estimated development time, and for having a debugging phase that seems to never end.

A strategy for mitigating risks in this kind of projects is to develop a carefully designed software system that can be fully debugged, tested and that could be reused in multiple projects, over and over. This subsystem thus becomes a high reliability foundation upon which we can build new applications by incrementally adding just the project-specific details. This kind of software subsystem is known as an enterprise framework. There are commercial application frameworks available for several domains, but none address the particular needs of automated manufacturing and test machines.

The design and application of object-oriented frameworks to produce reusable code was studied in early work by Johnson and Foote [13] and Fayad and Schmidt [16]. Fundamental ideas and methodologies for developing object-oriented frameworks have been studied by Johnson [15], [17], Roberts and Johnson [14], Codenie, et al. [18], Fayad, et al., Coplien and Schmidt [32], and Brugali et al. [20]. Experiences developing application frameworks are reported in Schmidt and Fayad [19], Fayad, Schmidt and Johnson [21], Fayad and Johnson [22] and Yassin and Fayad [23]. Fayad, et al. [4] make a strong case in favor of the pursuit of enterprise frameworks, describing their most important characteristics.

Brugali and Fayad [1] present an excellent overview of the state of the art in the field of software for robotics and automation, and argue that investing in enterprise frameworks make economic sense. They show that implementation scales of new systems based on enterprise frameworks may be compressed from years to months, or from months to weeks. This review presents a description of the three types of object-oriented frameworks: Enterprise Frameworks, System Infrastructure Frameworks, and Middleware Frameworks.

Raghavan and Waghmare [3] developed an object-oriented framework for distributed processing in industrial automation applications. Becker and Pereira [2] introduced an object-oriented framework called SIMOO-RT. Per definitions provided in [1], this system is better described as a middleware framework that provides an integrated development environment that covers the whole software development cycle: from requirements analysis, design, modeling, simulation and code generation. Projects implemented with SIMOO-RT, however, use fine-grained software components and requires intensive engineering work.

Wang and Shin [33] propose a software architecture for controlling automated industrial machines. Their model is based on Nested Finite State Machines (NFSM), which determine the behavior of the system’s software components. Behavior for these NFSM is generated from an executable formal specification. New applications are generated by constructing a mathematical model of the expected machine behavior. This model becomes the formal specification for their NFSM’s, which is done differently for each component and for each new application.

Although some of these systems [2], [3], [33] specifically target industrial automation applications, the problem of having to produce a new software system from the ground-up with every project still remains. In this paper we propose a solution to this problem.

III. SYSTEM ARCHITECTURE

The PEMSTAR Automation and Test Framework (PATF) system is structured in a layered architecture, as shown in Fig. 1. Each layer implements a different aspect of an industrial automation or test software system.
1) **Configuration Layer.** Launches the PATF engine and passes application-specific configuration information to the Engine layer.

2) **Engine Layer.** This is the core of the System Engine. Components in this layer implement an abstraction of the industrial automation and test domain, i.e. it provides most of the functionality that is common to this type of software applications.

3) **Application Layer.** Contains code components that provide custom functionality and features that are specific to the particular application at hand. It includes algorithms, GUI elements, database tables, data collection and data processing functions, for instance.

4) **Devices Layer.** Includes software components that represent the system’s hardware resources, like industrial robots, linear motion controllers, data acquisition boards, or instrumentation equipment.

Each of these layers is self-contained and bears specific responsibilities as part of the system. In **Configuration Layer**, only the PATF Configuration Settings change in order to specify the implementation of a particular project. **Engine Layer** abstracts our domain of interest and it does not change from project to project. **Application Layer** is the only one that, in general, will change from one application to the next, although its structure, interfaces and code templates remain the same. **Devices Layer** constitutes a set of pluggable, executable components that do not change from application to application, but the component mix may indeed change. **ActiveDevices** are, therefore, application-independent components that are reusable in executable form across projects.

Fig. 2 is a UML class diagram [24], [25], [26] showing a simplified view of the system’s class structure. The architectural structure throughout the system makes extensive use of design patterns [5], [7], [15], [17].

### A. Devices Layer

**ActiveDevices** play a central role in this design. The **ActiveDevice** specification [30] enables the implementation of software components that serve active objects [28] to control all sorts of instrumentation or actuator equipment. Each **ActiveDevice** runs in a separate process space and there is only one **ActiveDevice** object running for each physical industrial automation device installed. Since there could be multiple Application Sequences running in concurrent mode at any time and might need random access to a certain peripheral device, it becomes necessary to control mutually exclusive access to each **ActiveDevice**. For that purpose, each **ActiveDevice** provides its own mutex object [8], [9], which is made available for clients to grab in order to gain access to its services.

This layer has two types of components: **ActiveDevice** and **ActiveDevice_Container**. There is only one instance of the **ActiveDevice_Container**. On the other hand, there can be any number of **ActiveDevice** objects in the container and multiple instances of the same **ActiveDevice** class are allowed, as long as each instance refers only to a single physical device. For example, a 3-axis Cartesian robot would require three instances of a servo motor controller **ActiveDevice** component.

Regardless of the kind of device being controlled, all components of data type **ActiveDevice** implement the standardized **ActiveDevice** specification. This specification comprises an object interface and a given state machine behavior and protocol. Table 1 and Table 2 list some of the most important Attributes and Operations, respectively, in the standardized interface. **ActiveDevice** components extend this basic interface with device-specific attributes and operations. For example, a robot **ActiveDevice** component might have additional attributes like: Force(), RobotReady(), or EndEffectorAttached(), or operations like: Move(nX, nY, nZ), SetSpeed(nNewSpeed), or CurrentPosition(nX*, nY*, nZ*).

**Figure 3** shows an UML state diagram describing the top-level **ActiveDevice** state machine behavior. This standard behavior must also be implemented by all **ActiveDevice** components, regardless of the kind of physical device they are controlling.

### B. Application Layer

This layer contains the application-specific code, which defines the application’s algorithms, functionality, and customization of the System Engine (described in Section **Engine Layer** below). There are several major components in this layer: **App_Sequence**, **Subject_Observer**, and **GUI_Panels**. Each of these components is compiled independently and may be tested and run either in isolation or using stubs. Let us review each of these components in turn.

1) **App_Sequence.**

The central element of this layer is the **App_Sequence** component. This is a collection of classes that define two types of sequences: **State Sequence** and **Application Sequence**. State Sequences define the custom behavior of the System Engine that is required for the application at hand.

The system as a whole behaves as a finite state machine, as described in Section **System Dynamics**. For the moment, suffice it to say that there are a total of eight states: **INITIALIZE**, **SHUTDOWN**, **MANUAL_SETUP**, **MANUAL_OPS**, **MANUAL_IDLE**, **AUTO_PRODUCTION**, **AUTO_SEQUENCE**, and **AUTO_IDLE**.
State Sequences define the names and order of execution of the Application Sequence classes scheduled to run in each state. Application Sequences are application-specific tasks that encapsulate algorithms for building or testing products, or implement mathematical computation algorithms.

In general, State Sequences run only once when the system switches to a new state, and only one of these objects is active at any given time. On the other hand, Application Sequences may run concurrently and multiple instances of each class may be created. State Sequences implement an interface with a single attribute: ProcessList, and a single operation: Execute(). ProcessList holds an array of strings with the names of processes available in that state. A process is an ordered list that includes one or more Application Sequences, which may run in any combination of Sequential and Concurrent tasks. Operation Execute() specifies the class name and mode of execution for each task. If a class is specified to run in sequential mode, the Application Sequence is executed to completion before instantiating the next class in the list. If a class is specified to run in concurrent mode, it is instantiated and immediately after being launched, the next class in the list is also instantiated.

The purpose of Application Sequences is to separate concerns, and thus, to make software easier to understand, manage, evolve, test and debug. Application Sequences cooperate to perform complex tasks, communicating and synchronizing with each other by exchanging SyncPoint objects. This operation is commonly known as rendezvous [9]. The engine Scheduler manages this operation, as described in Section IV.

The SyncPoint object carries information about the synchronization point identifier, partner’s name and exchange data, which may be converted by the sequence to any data type in order to decode information that may define a subsequent execution path.

Application Sequences are structured in execution steps. A class of this type may have an arbitrary number of steps, with special cases Step 0, which terminates execution, and Step 1 that declares all synchronization points that shall be called during execution of the algorithm. Application Sequence objects hold references to other objects that are available within the system at run-time, including GUI_Panels and the ActiveDevice_Container. Each Application Sequence object is assigned a unique Client ID, which is used for accessing shared resources. The full dynamics of these relationships is detailed in Section IV.

2) GUI_Panels.

The System Engine provides a general-purpose Graphical User Interface (GUI) that captures the most common features required in industrial automation systems. It is not possible, however, to produce a GUI that will encompass every possible function that any user will ever need.

For this reason, we included a set of customizable objects, which in turn can contain other objects and custom code, allowing application programmers to customize their applications without too much overhead. There can be up to eight GUI Panel objects that are served at run-time by the GUI_Panels component. These objects work with the Main_GUI component (described in Section C. Engine Layer) through an object composition mechanism that allows them to be compiled separately and independently, and still be integrated at run-time such that their separation is transparent to the user.

GUI Panels can be used for any purpose. Typical applications include: product counters, statistical quality data, statistical process control, graphs, data-bound table displays, and manufacturing or test progress information.

3) Subject_Observer.

This component is based on the Observer design pattern [5], also known as Publish-Subscribe and serves two basic object types: a general-purpose Subject object and one or more Observer objects. Subject objects are all instantiated from the same class, regardless of the particular application. Application Sequences have access to Subject objects, each of which holds a different type of information, like database recordsets, test results, statistical data, and so forth. Programmers working on Application Sequences need only to worry about writing data to Subject objects and do not need to know what happens to it afterwards.

Objects of types Subject and Observer run in a separate process in the background, without adding overhead to the main application. As described in the Observer design pattern definition, Subjects notify Observers when a new update is available. Please refer to [5] for more details on this mechanism. Observers are customizable objects that typically write data to GUI_Panels, generate database records, produce graphs, charts or tables, calculate statistics, or any other automated updating task.

System Engine expects all pluggable objects to be compliant with the specified interfaces and protocols, in addition to file location within the framework file system, effectively forcing application programmers to meet the expected interface specification, using code templates that are provided as part of PATF. Application Layer is the “white-box” part of this framework.

C. Engine and Configuration Layers

Upon startup, the PATF Launcher instantiates and customizes the System Engine passing a PATF_Config object carrying the necessary information, and then hands control over to the Scheduler object, effectively starting Inversion of Control.

The Engine Layer implements a general-purpose platform for concurrent and serial execution of custom algorithms. In contrast to Application Layer, the Engine Layer is the “black-box” part of this framework, since its components do not need to be changed by programmers, regardless of the application being implemented. This layer is also called the System Engine. The engine provides a number of services,
functionality and behavior that have been identified to be common to all applications in the industrial automation and test domain.

From a high level viewpoint, the system behaves as a finite state machine, as described in Section IV - System Dynamics. Transitions between states are, therefore, predefined for all applications. Nevertheless, as described in Section III-B above, these states can be customized with algorithms embedded in State Sequence classes. The most important engine components are described below.

1) Scheduler
The role of the Scheduler is similar to that of an orchestra conductor. This component is responsible for organizing, instantiating, managing and destroying objects, and for launching, synchronizing and controlling concurrent tasks. There is only one Scheduler, and it is the one object in charge of running the whole system.

Scheduler is the object that is instantiated by the CustomApp program upon start, and it takes over execution control from that point forward. Scheduler is the parent object to most other objects in the system, including App_Sequence, ActiveDevice_Container, GUI_Panels, Main_GUI and multiple Processors.

2) Processor
This component serves active objects of type Processor. These objects are part of a larger composition similar to the Command Processor design pattern [7]. During initialization, a Processor is given a task to perform, i.e. a command to process. A command is an Application Sequence class name that the new Processor instantiates, see for example, the Strategy design pattern [5].

Application Sequence classes are, therefore, completely isolated from the framework engine, remaining unaware of the inner workings of the underlying system, which simplifies the complexity of their code, and subsequently, the application programmer’s job.

3) Main_GUI
The user interacts with the system through the services of the Main_GUI component or those of GUI_Panels. Main_GUI implements the system’s finite state machine described in Section IV below, working very similarly to the State design pattern [5]. Main_GUI maintains relationships with Scheduler and GUI_Panels, and instantiates an inner User_Management object, described below. The Main_GUI features and behavior is a generalization of what is typically found to be required in an industrial automation system, including facilities to initialize the system, select and launch individual processes, cancel processes, access to hardware configuration screens (ActiveDevice_Container), product load and unload, and production or manual run options.

4) User_Management
Similar to GUI_Panels, this component also integrates seamlessly with the Main_GUI graphical interface at run-time. The user access validation services provided by the User_Management component, operate within the scope of Main_GUI only. Once a user’s request has been authenticated, a message requesting the operation is dispatched to Scheduler, removing this burden from a centralized controller.

IV. SYSTEM DYNAMICS

The proposed domain abstraction classifies operation modes of automated machines in either Manual Mode or Automatic Mode. Under these two operating modes, there are six system states, in addition to INITIALIZE and SHUTDOWN. Fig. 4 shows these states and the events that cause transitions between them, while Fig. 5 shows the collaboration among objects at run time.

1) Manual Mode
This mode is used to modify operational parameters and for performing manually controlled operations like jogging a robot, for example. Safety devices are of utmost importance in industrial automation machines. They play a critical role in preventing operators from getting injured. In manual mode, however, these safety devices are overridden to enable a qualified technician to perform certain operations that require full control over the machine, as is the case of machine troubleshooting or other maintenance operations. This unsafe operation mode is hard-coded into the system’s engine. Before entering this mode, users are authenticated and only allowed if they hold the proper credentials, as determined by the User_Management component. All safety-device signals are ignored and the machine is allowed to run, although displaying a warning message on the user interface through the Main_GUI component.

2) Automatic Mode
AUTO_MODE is the normal operation mode for this type of machines. In contrast to manual mode, all system devices and subsystems are engaged and a built-in safety monitoring system watches for incoming alarms from input/output signals configured as safety devices and automatically calls an emergency shutdown procedure when flagged. Automatic mode also allows users to perform maintenance operations where full speed and safety-device enforcement are required, for example, robot coordinate system calibration or product load and unload.

3) Change of State
System control is centralized in the system Scheduler, as described above. Consequently, Scheduler handles all state-change requests that are generated within any component across the system. Requests may be raised by direct user intervention, or as a result of program execution when handling an exception or an error condition. Child objects send messages to Scheduler containing information about the
request and Scheduler reacts validating the request and then proceeds to change the system’s state, synchronizing all subordinate components afterwards.

4) **Manual_Idle and Auto_Idle States**

Upon completion of the initialization routines, the system goes directly to the Auto_Idle state. Idle states are safe states, i.e. the machine will not move or otherwise start a free-running operation. The most important operations in these states are user authentication and requests for validation before authorizing administrative operations or state transitions.

5) **Manual_Setup State**

Transition to this state causes the ActiveDevice container screen to be displayed automatically. Here, the authorized user may control and configure each ActiveDevice component individually, modify operational parameters, configuration, or troubleshooting.

6) **Manual_Ops State**

In the manual operations state, technicians may run production sequences in a step-by-step fashion for debugging manufacturing processes. Other sequences may be written to accomplish other goals as well. Execution, however, may be restricted to reduced speed or limited actuator force.

7) **Auto_Sequence and Auto_Production States**

Operationally, these two states are identical, but are semantically different, for the sake of clarity to the user. The difference is that in Auto_Production state, the machine outputs physical products, either manufactured or tested, whereas in Auto_Sequence state the machine runs as if it were manufacturing or testing actual products, but some product sensors may be overridden to fine-tune the manufacturing process, perform a dynamic calibration or other similar purposes. All safety devices remain engaged at all times and cannot be overridden.

8) **Task Launch and Processing**

Whenever Scheduler changes state, it executes the application-specific tasks for that state, as defined in State Sequence classes, shown in the UML class diagram in Fig. 6. The CurrentProcess attribute holds the name of the algorithm that is currently assigned to the Execute() operation. See, for reference, the State design pattern [5]. For each step (nStep) in the execution, this operation returns the name of the class to be instantiated next and its corresponding execution mode; which may be either MODE_CONCURRENT or MODE_SEQUENTIAL.

Sequential sequences are launched and executed to completion before advancing to the next step in the state algorithm. For concurrent sequences, Scheduler instantiates and launches the specified class, immediately returning to fetch the next step in the state algorithm, while the concurrent sequence is being executed in a separate thread. The following code fragment illustrates the structure of the Execute() operation:

```c
switch (nStep) {
    case 1:
        nIndex = 1
        nExecMode = MODE_CONCURRENT;
        chClassName = \App_SequenceOne;
        break;
    case 2:
        nIndex = 1;
        nExecMode = MODE_CONCURRENT;
        chClassName = \App_SequenceTwo';
        break;
}
```

Parameter nIndex enables application programmers to create more than one instance of a given class at a time, generating a unique identifier for each new process. Scheduler creates a new instance of the Processor class for each process to be launched, as requested by the current state algorithm. During initialization of Processor objects, these are given references to the system’s resources, including the parameter chClassName returned by the state class’s Execute() operation, which determines the specific algorithm that Processors are to run.

9) **Task Execution**

The new Processor object instantiates the requested class and executes the corresponding algorithm. See, for reference, the Command Processor design pattern [7]. The object instantiated by Processor is of type App_Seq (see Fig. 6). As described earlier, each of the classes of this type contains an application-specific algorithm that addresses an individual aspect of the system. In the example above, for instance, App_Sequencetwo class could define an algorithm for product-handling operations in a robotic assembly cell.

Application sequences implement separation of concerns at the application programmer level. Applications are built by writing independent algorithms that are concerned with different aspects of the application at hand. For example, several sequence classes for each robot task; another class for controlling measurement instruments; a separate class for a part feeder, and yet another class for performing mathematical calculations. Only the system’s resources limit the number of simultaneous tasks that can be executed. Objects instantiated from these classes become separate, independent processes, that are capable of synchronizing their execution and sharing information between them, as described in the next section.

Application sequences specify an algorithm for a given task, and this algorithm is executed by calling the class’s RunSequence() operation, which takes an argument nStep and returns an integer value, indicating the requested step for the next cycle. Execution is performed as a series of discrete steps structured as a single switch statement that branches execution according to the current value of nStep. Argument
nStep may take any integer value and, thus, allows the application sequence class programmer to loop or jump to any step within the structure as desired. For convenience, steps are usually defined in an enumeration. For example,

```c
enum MySteps {
    Step1 = 2,
    Step2,
    JumpHere1,
    Loop
};
```

Legal values for executable steps are:

```c
{nStep >= 2}
```

Because there are certain special values for nStep. These are summarized in Table 4.

Separation of concerns at the custom application level, as achieved with this approach, is a major advantage because it enables programmers to tackle very complex automation projects by concentrating on individual aspects of the system at a time.

10) Task Synchronization (Rendezvous)

Each task runs in a separate thread, but these threads may synchronize their execution by performing a rendezvous operation at a predetermined stage of their individual execution algorithms [8], [9]. Rendezvous is realized by sending a synchronization object to Scheduler, which is responsible for matching rendezvous partners.

Table 5 shows the synchronization object structure. Two synchronization objects, or SyncPoint, define a synchronization point. Tasks that are partners in a rendezvous operation supply one SyncPoint each. For rendezvous to occur, both SyncPoints must hold equal values in their respective SyncName properties, and their PartnerName property must hold the object name of the other party. Scheduler compares SyncName, SeqName and PartnerName properties to determine a match. If the second partner is not ready for rendezvous, the partner arriving first gets suspended until the other party is ready. Upon synchronization, the contents of their respective SyncData properties are swapped between the partners. Once the rendezvous operation has been completed, application sequences may cast the character string held in the SyncData property onto a different, more convenient data type in order to process the information and react accordingly.

Therefore, rendezvous is a synchronization and indirect communication mechanism for information exchange between fellow application sequences, or tasks. A Petri net is a useful tool for modeling rendezvous and concurrent execution [27].

11) Interprocess Communication

Communication between processes also occurs behind the scenes. While rendezvous transfers information synchronously from one App_Sequence object to another via the system Scheduler, all other software components exchange information with Scheduler in an asynchronous fashion by calling their Message operation.

A third communication mechanism consists of sending control messages from the system Scheduler to Processor objects, which may be done via a one-to-one message, or a system-level broadcast. The purpose of these control messages is to inform Processors about controlled shutdown procedures, execution errors, emergency shutdowns, et cetera.

12) Error Conditions

The PATF system is a dynamic collection of active objects that are integrated into a single, orchestrated system at runtime. Nevertheless, violation of built-in policies or run-time errors may occur in any component, and abnormal situation alarms must be propagated. These conditions are managed by the system Scheduler, which determines how to proceed, and sends an appropriate control message to the ProcessController object.

13) Debugging tools

Debugging multithreaded programs is often challenging. For this reason, the PATF System includes a set of built-in debugging tools by design. These tools include an automatic time-stamped log tracing mechanism that generates separate log files for each thread. These may be later combined as an aid in the analysis of concurrent tasks. System status and executed commands may also be displayed on the screen in real time with three levels of detail. The most detailed level provides information of every operation executed by each component and each App_Sequence object being executed.

V. Case Study

In order to test the concepts put forth in the proposed architecture, a component-based, object-oriented application framework was implemented based on this design, and tested on three different applications within the field of automated industrial manufacturing and testing.

The first application, System A, is an automated assembly cell for building Micro Electro-Mechanical System (MEMS) devices. This task requires very high precision, high performance and high flexibility. It includes a vision-guided industrial robot, multiple servo-controlled linear actuators and a range of other industrial automation devices.

The second application, System B, uses the same type of robot in a completely different cell platform for automatically testing the same MEMS devices. In this case, the system features multiple part feeders to enable uninterrupted operation. This system uses high-precision optical and instrumentation equipment for testing products and performs a series of mathematically intensive calculations.

The third application, System C, is a manual version of the robotic tester, using the same optical equipment and mathematical algorithms, but manual part handling, interactive operation with the user and a different set of operator safety mechanisms.
policies.

1) Implementation

The system was implemented to run on a Microsoft Windows® platform using COM technology. Active objects were implemented as multithreaded COM servers [29].

In order to simplify the use of ActiveDevices, these components were implemented in two parts: an ActiveX control and an ActiveX EXE. The ActiveX control can be dragged and dropped onto an object container, which in turn instantiates an out-of-process COM server in a way that results transparent to the application programmer.

Fig. 7 shows an UML Deployment Diagram with the code components required to implement System A, described above. Similar structures are required to implement Systems B and C.

Software components represented with white symbols are reusable in binary (compiled) form across multiple applications, and shaded symbols represent components that must be modified with application-specific code. Therefore, the set of software components that comprise an instance of system architecture described herein is divided in two subsets according to their relation to the custom application: Independent and Dependent.

2) Application-independent components

Components represented in the UML Deployment Diagram in Fig. 7 are directly mapped from high-level classes in the System Architecture shown in Fig. 2. Please refer also to Fig. 1, and notice that most components in the Devices Layer are reusable in executable form. The only exception is ActiveDevice_Container, which is discussed below. All of the components in the Engine Layer are reusable in binary form across multiple applications.

From an economical point of view, components that are reusable in binary form are very valuable assets for an organization, since they are designed, developed, and tested once, and can then be used in many future applications using the same devices abstracted by these software components.

3) Application-dependent components

Naturally, most of the components in the application layer must be customized to fit the target application. Nevertheless, these components are developed based on code templates, where more than 60% of the source code is reused, with the exception of App_Sequence, which only reuses a code skeleton, or in other words, the interface and code structure.

Going back to ActiveDevice_Container, we said that there is practically no code required in this component, since its only purpose is to package the selected ActiveDevice components together into a single, compiled component.

As its name hints, Subject_Observer serves objects of two types: Subject and Observer. There is only one class for Subject, which requires no customization whatever. It is reused at the source code level as-is. On the other hand, an application-specific Observer is implemented by deriving it from the Observer base class. A typical derived Observer class requires modifying or writing less than 100 lines of code.

The case for GUI_Panels is very similar, although most of the modifications are related to User Interface objects, like command buttons, data-grids, data-bound controls, labels, and so forth.

Finally, we get to the executable program CustomApp, which in fact, is only a generic name for this component. This program has the only task of instantiating the System Scheduler with the main settings for the user interface and system configuration. This is a template project that requires customizing less than 10 lines of code. In practice, this program is usually compiled with the application name, for instance, RobotCell.exe. Once instantiated, the system Scheduler takes over, and from that point on, CustomApp is actually relegated to the background.

VI. Evaluation

To evaluate the impact of this software architecture on code reuse and programmer productivity, we start by considering the enterprise application framework described in Section V as an instance of the software architecture and design presented in Sections III and IV.

With the intention of being objective, we surveyed the software metrics literature, but were unable to find an appropriate, all-encompassing metric suite that would measure the benefits derived from using enterprise frameworks, where reuse includes both design and executable code. Most proposed metrics address class structure, complexity and static relations that use source code files and other fine-grain elements as inputs. The coarse-grain modularity and functionality of an enterprise framework lay beyond the scope of such metrics. We found that new metrics are needed to address the influence of reuse design, architecture, and other system features located at higher levels of abstraction.

Chidamber and Kemerer [34] proposed a metrics suite for object-oriented design (C&K Metrics) that concentrates on measuring class complexity and static structure of the software architecture. These metrics provide indicators about the quality of a design, but the only metric that relates to reuse is Depth of Inheritance Tree (DIT). However, if we consider that our approach favors run-time object composition rather than inheritance to achieve software subsystem reuse, it was determined that these metrics did not yield the information we needed.

Other authors have proposed different metrics, some of them specifically targeting software reuse, but all of them focus on fine-grained software entities like functions, classes or reusable libraries [35], [38], [39]. Others measure reusability of isolated low-level software components [36], or commercial code components like Java Beans [37], but still, we found these metrics unsuitable for the kind of reuse obtained with enterprise frameworks.
Cardino et al. [41], realized the special measurement needs of object-oriented frameworks and they present an outline of the features affecting their reusability, but they do not suggest a specific set of objective metrics. Work done at AT&T Labs during the past decade [38], [39], resulted in an interesting study focusing on the quantification of software reuse in large industrial projects [40]. Unfortunately, although these systems reuse large general-purpose function libraries, their metrics do not address the reuse of software architecture and design, focusing on the development phase only.

In the case of application frameworks, we think that reuse of architectural design is a major factor in the success of deployed applications using it. When architectural design is reused, the design phase in the software development cycle is simplified and reduced to mapping the application’s specific requirements to the different components of the standardized architecture, resulting in a well-known system organization that streamlines the implementation phase as well. This effect includes important engineering labor savings, which should be quantified in order to take this important benefit into account.

A. Methodology

Given this situation, we decided to take a more pragmatic approach to measuring the impact of framework reuse, and carried out a semi-objective evaluation by using historical data from comparable projects done previously.

Table 6 shows a list of projects developed at PEMSTAR and their respective software development cost in man-hours. All listed applications are very similar robotic cells for automated manufacturing. They all integrate the same robot, the same machine vision system, the same servomotors for linear motion and other industrial automation tools. Their main difference is that each machine builds a different, high-precision product. Projects A, B, C, and D were developed using the traditional approach to software development. Project E was implemented using the PATF framework. With this information as our starting point, we took the number of engineering-hours that were required to design, implement, test and debug previous projects and then compared it to the effort required to implement a new application using the PATF subsystem.

B. Results

Let us now characterize this architecture and its reusable code, as fully implemented in the PATF enterprise framework.

1) Code Reusability

Table 7 shows the PATF Black-Box Reuse Profile, which lists software components that are reused in binary (executable) form. The total number of Shipped Source Instruction lines (SSI) is given as an indicator of the component’s size and cost. This is later used as a reference point when quantifying labor savings. Since these components are reused without any changes, the total black-box reuse ratio is 100%.

Table 8 shows the PATF White-Box Reuse Profile, which represents the source code reuse ratio for components reused as code templates, which have to be modified to suit the new application at hand. Naturally, all application-specific code components show low source code reuse ratios. Nevertheless, white-box reuse ratio reaches 30%, which accounts mostly for abstract classes and other PATF interface elements.

Table 9 summarizes black-box and white-box reuse ratios, yielding a net shipped code reuse of almost 90%.

2) Cost Savings

It is evident that a reduction in the total implementation effort translates directly into cost savings. Looking at the previous projects, we find that the median effort for Projects A, B, C, and D is 2,258 man-hours, and the average is 2,269 man-hours.

On the other hand, the total cost for the design, implementation and test of the PATF framework was approximately 2,500 man-hours. As of this writing, ten different applications have been implemented based on the PATF framework, yielding a prorated cost of approximately 250 man-hours per system.

If we add this PATF unit cost to the implementation effort for Project D, and take the median as a representative cost value for projects implemented using the traditional approach, we get the results shown in Table 10. The numbers show that projects implemented with the PATF subsystem cost 1/3 of what it would cost if they were developed from scratch. Likewise, since our cost units are man-hours, the same numbers show that the software team gets the job done three times faster than with the traditional approach.

Savings in engineering cost is an important piece of information for justifying the greater investment required to develop software that is reusable by design.

3) Other Observations

Since PATF implies that a complete set of design decisions have been made for software developers, they did not have to spend any time at all designing their new applications. Likewise, developers had to spend no time thinking about how to identify, create or abstract new modules or interfaces, since all of them are fixed in the framework. Functional policies that are typical to most applications within our particular domain are also included in the underlying design. Therefore, the only task left for them to do was to focus on the particular functional details that made the target application unique, i.e. the specific algorithms of their new assignment.

Due to the phenomenon of inversion of control, programmers are forced to write application-specific code strictly following the guidelines and interfaces prescribed in the PATF design. The resulting code was more homogeneous, standardized and consistent across the team, as compared to code from previous projects.

We also found that a large proportion of the final code in a deployed application is reused in binary form across multiple
VII. Conclusions

In accordance to previous reports [4], [16], [19], [23], our results are very positive, and we have achieved an estimated level of executable code reusability of about 90% and cost savings of about 70%. Productivity of software development teams has also seen improvement, though the learning curve has been slow for some programmers. Nevertheless, the overall results are encouraging. After this study was completed, the PATF system has been deployed in five additional applications with similar success.

The main contribution of this paper is the proposal of a generalized software architecture for the industrial automation and test domain that fosters a systematic development of standardized application software. This approach has proved to have a positive impact on programmer productivity in this domain. The described System Engine and ActiveDevice specification enable binary code reuse, and promote the construction of reusable software assets. An additional benefit of applying the PATF approach is the production of standardized systems that are more consistent and robust from application to application.

VIII. Future Work

We found that there is a need for more adequate metrics that address the particular characteristics of enterprise frameworks as higher-level software entities. We make a call for researchers to address this issue, while we plan to investigate this topic further ourselves.

It is clear that the architecture presented in this paper can easily be extended to operate in a multi-processor platform by distributing Processor objects over a network. This enhancement will enable powerful parallel processing for computationally intensive applications like real-time, multi-camera image processing, for instance. This enhancement is currently under development at PEMSTAR.

Acknowledgment

I would like to express my appreciation to M. Troutman and R. Ahmann for their support of this project; to R. Real and G. Flynn for their contributions and many productive discussions, and to the PEMSTAR programming team, especially L. Yan and R. Chowdhury, for their crucial help during the implementation of this system.

References


Carlos O. Morales (M’00) is Senior Systems Architect with PEMSTAR, Rochester, MN, USA, where he leads engineering teams developing advanced industrial automation and test systems. He works in the areas of machine vision, intelligent agents, and software architecture. His current research interest is distributed intelligent agents.

Mr. Morales holds BS degrees in physics and electrical engineering and a MS degree in computer engineering. He is a member of IEEE, ACM and SPIE and has been a speaker and instructor for IEEE, SPIE and industrial associations at numerous international technical conferences in North America and Europe. He was the winner of the 2001 Imaging Solution of the Year award for his work on Machine Vision, and was a research fellow with the National Science and Technology Council of Mexico in 1994 and 1995.
Figures

Fig. 1. Architecture of the PEMSTAR Automation and Test Framework (PATF). Configuration and Application layers encapsulate the application-specific code, which is compliant with standardized code templates and interfaces. The other two layers, Engine and Devices, comprise the general-purpose multithreaded execution platform. Shaded boxes indicate template-based customizable components. The rest are software components that can be reused in binary (executable) form.

Fig. 2. PATF System, simplified class diagram. This UML class diagram shows the most important classes, their relations and the static structure of the proposed system.
Fig. 3. ActiveDevice State Diagram. This UML state diagram shows the top-level view of the behavior that all ActiveDevice components must implement in order to comply with the specification. This is the standard behavior expected by the framework engine.

Fig. 4. System State Diagram. This UML state diagram shows the top-level view of the system’s behavior. Certain components follow the current system state, like GUI and Scheduler, and others are state-independent, like Subject-Observer, GUI Panels and Processors.
Fig. 5. UML Collaboration Diagram showing a typical dynamic interaction between PATF objects at run time. In this simplified example, three concurrent processes are executing one instance of algorithm App_Sequence_A and two instances of algorithm App_Sequence_B. Objects served by the Subject_Observer components are shared resources to which all processes have access.

Fig. 6. PATF App_Sequence. This UML class diagram shows the App_Sequence software component class structure. This organization allows programmers implementing new applications to focus on individual aspects of an application at a time, or to organize a programming team to work in parallel. Each class implements a single task within the system.
Fig. 7. System A Deployment Diagram. This UML deployment diagram shows all the implemented code components and their relationships. Shaded boxes represent application-specific code. White boxes represent components that are reused in binary (compiled) form.
TABLE I  
ACTIVEDEVICE OBJECT INTERFACE (ATTRIBUTES)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentClientID</td>
<td>Client ID number to verify access rights.</td>
</tr>
<tr>
<td>ClientMessage</td>
<td>String used to communicate information to client.</td>
</tr>
<tr>
<td>ClientResponse</td>
<td>String encoding information from client.</td>
</tr>
<tr>
<td>ErrorState</td>
<td>Error condition code.</td>
</tr>
</tbody>
</table>

Common interface implemented by all ActiveDevice components. In addition to these basic interface elements, it can be extended to include additional attributes and operations that are specific to the device being implemented.

TABLE 2  
ACTIVEDEVICE OBJECT INTERFACE (OPERATIONS)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LockDevice</td>
<td>Returns the object’s mutex when it becomes available.</td>
</tr>
<tr>
<td>ReleaseDevice</td>
<td>Releases the object’s mutex.</td>
</tr>
<tr>
<td>Initialize</td>
<td>Runs any initialization procedure that might be required by the device being controlled.</td>
</tr>
<tr>
<td>Calibrate</td>
<td>Performs any calibration procedure that might be required by the device being controlled.</td>
</tr>
<tr>
<td>Run</td>
<td>Sets the device in run mode.</td>
</tr>
<tr>
<td>AlarmReset</td>
<td>Resets the ActiveDevice to continue normal operation after an error condition.</td>
</tr>
</tbody>
</table>

In order to ensure system consistency, all operations include the calling client ID as a parameter, in addition to any other parameters that the operation call may require. This way, ActiveDevices enforce a single-client access policy that enables only the current client to modify the ActiveDevice state. If an ActiveDevice object receives a call from a client other than the currently authorized, the call is disregarded.
### TABLE 3
**ACTIVEDEVICE CONTAINER INTERFACE**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchedulerRef</td>
<td>Reference to the Scheduler object.</td>
</tr>
<tr>
<td>ActiveDeviceForm</td>
<td>Reference to object-container holding ActiveDevice components.</td>
</tr>
<tr>
<td>SetupShow</td>
<td>Displays the ActiveDevice container GUI.</td>
</tr>
<tr>
<td>SetupHide</td>
<td>Closes the ActiveDevice container GUI.</td>
</tr>
</tbody>
</table>

ActiveDeviceForm is a MS-Windows form object that is used as an object container. ActiveDevice components are added to a project by dragging and dropping ActiveX controls onto the form.

### TABLE 4
**APPLICATION SEQUENCE EXECUTION STEPS**

<table>
<thead>
<tr>
<th>Step Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><em>Finished execution.</em> This value is returned when the algorithm terminates normally.</td>
</tr>
<tr>
<td>1</td>
<td><em>Rendezvous declarations.</em> Step 1 is reserved for the publication of synchronization points (rendezvous) used within the algorithm.</td>
</tr>
<tr>
<td>-1</td>
<td><em>Unexpected shutdown requested.</em> Aborts execution of all concurrent processes presently running, in a controlled fashion. Shutdown is requested due to a run-time error generated within the sequence. An informative character string about the error is stored in the object’s Message property.</td>
</tr>
<tr>
<td>-2</td>
<td><em>Emergency shutdown requested.</em> Immediately stops execution of all concurrent processes presently running. Shutdown is requested due to a critical error generated within the sequence. An informative character string about the error is stored in the object’s Message property.</td>
</tr>
</tbody>
</table>

Reserved return values for parameter nStep in the RunSequence() operation of application sequences. When errors are found, the application sequence sets a negative value, according to severity. Execution of step 1 is part of the normal initialization of Processor objects.
TABLE 5
SYNCHRONIZATION OBJECT STRUCTURE (SyncPoint)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncName</td>
<td>Synchronization point name.</td>
</tr>
<tr>
<td>SeqName</td>
<td>Caller sequence’s name.</td>
</tr>
<tr>
<td>ClientID</td>
<td>Caller sequence’s unique client ID.</td>
</tr>
<tr>
<td>PartnerName</td>
<td>Synchronization partner sequence name.</td>
</tr>
<tr>
<td>SyncStep</td>
<td>Execution step at which current synchronization instance must be executed.</td>
</tr>
<tr>
<td>SyncData</td>
<td>Character string holding data to be sent to synchronization partner. After</td>
</tr>
<tr>
<td></td>
<td>synchronization, this property holds the information received from partner.</td>
</tr>
</tbody>
</table>

Application sequence tasks executing in concurrent mode may send synchronization objects to perform a rendezvous operation at a predetermined execution step. During rendezvous, sequence objects may exchange information via the SyncData property.

TABLE 6
TOTAL SOFTWARE DEVELOPMENT EFFORT

<table>
<thead>
<tr>
<th>Projects with Similar Applications</th>
<th>Total Engineering Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project A</td>
<td>3,030</td>
</tr>
<tr>
<td>Project B</td>
<td>2,290</td>
</tr>
<tr>
<td>Project C</td>
<td>2,225</td>
</tr>
<tr>
<td>Project D</td>
<td>1,530</td>
</tr>
<tr>
<td>Project E (using PATF)</td>
<td>384</td>
</tr>
</tbody>
</table>

Total engineering effort (in man-hours) required for producing closely comparable software for assembly machines in previous projects. No code was reused among Projects A, B, C, and D. Project E was implemented using the PATF framework. All applications are robotic assembly cells assisted with machine vision for building high-precision products.

TABLE 7
PATF BLACK-BOX REUSE (PROJECT E)

<table>
<thead>
<tr>
<th>Component</th>
<th>SSI</th>
<th>RSI</th>
<th>Reuse %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>12,963</td>
<td>12,963</td>
<td>100%</td>
</tr>
<tr>
<td>Processor</td>
<td>5,961</td>
<td>5,961</td>
<td>100%</td>
</tr>
<tr>
<td>Main GUI</td>
<td>8,046</td>
<td>8,046</td>
<td>100%</td>
</tr>
<tr>
<td>User Management</td>
<td>4,584</td>
<td>4,584</td>
<td>100%</td>
</tr>
<tr>
<td>ActiveDevice Robot</td>
<td>10,864</td>
<td>10,864</td>
<td>100%</td>
</tr>
<tr>
<td>ActiveDevice DIO</td>
<td>2,449</td>
<td>2,449</td>
<td>100%</td>
</tr>
<tr>
<td>ActiveDevice DMM</td>
<td>2,170</td>
<td>2,170</td>
<td>100%</td>
</tr>
<tr>
<td>ActiveDevice Vision</td>
<td>4,762</td>
<td>4,762</td>
<td>100%</td>
</tr>
<tr>
<td>ActiveDevice DeviceNet</td>
<td>6,842</td>
<td>6,842</td>
<td>100%</td>
</tr>
<tr>
<td>ActiveDevice Servo</td>
<td>5,308</td>
<td>5,308</td>
<td>100%</td>
</tr>
</tbody>
</table>

TOTAL 63,949 63,949 100%

PATF Reuse Profile. In Black-Box reuse, software components are reused in their compiled (executable) form. Source Instructions are given as an indirect indicator of the development effort required to produce these components.

Key: SSI = Shipped Source Instructions, RSI = Reused Source Instructions.
TABLE 8

<table>
<thead>
<tr>
<th>Component</th>
<th>SSI</th>
<th>RSI</th>
<th>Reuse %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>597</td>
<td>597</td>
<td>100%</td>
</tr>
<tr>
<td>CustomApp</td>
<td>354</td>
<td>320</td>
<td>90%</td>
</tr>
<tr>
<td>ActiveDevice Container</td>
<td>288</td>
<td>225</td>
<td>78%</td>
</tr>
<tr>
<td>GUI Panels</td>
<td>1,761</td>
<td>663</td>
<td>38%</td>
</tr>
<tr>
<td>AppSequence</td>
<td>6,620</td>
<td>1,421</td>
<td>21%</td>
</tr>
<tr>
<td>Observer</td>
<td>1,678</td>
<td>166</td>
<td>10%</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>11,298</td>
<td>3,392</td>
<td>30%</td>
</tr>
</tbody>
</table>

PATF Reuse Profile. In White-Box reuse, software components are reused as code and project templates. These templates include all abstract classes defining the component interfaces. SSI and RSI are counted when a new application using PATF is released.

Key: SSI = Shipped Source Instructions, RSI = Reused Source Instructions.

TABLE 9

<table>
<thead>
<tr>
<th>Component</th>
<th>SSI</th>
<th>RSI</th>
<th>Reuse %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Box Reuse</td>
<td>63,949</td>
<td>63,949</td>
<td>100%</td>
</tr>
<tr>
<td>White-Box Reuse</td>
<td>11,298</td>
<td>3,392</td>
<td>30%</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>75,247</td>
<td>67,341</td>
<td>89%</td>
</tr>
</tbody>
</table>

Total source code reuse in an application implemented with the PEMSTAR Automation and Test Framework.

Key: SSI = Shipped Source Instructions, RSI = Reused Source Instructions.

TABLE 10

<table>
<thead>
<tr>
<th>Software Implementation Technique</th>
<th>Total Engineering Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total project cost using traditional approach (median)</td>
<td>2,258</td>
</tr>
<tr>
<td>Total project cost using PATF</td>
<td>634</td>
</tr>
</tbody>
</table>

Development Cost Reduction: 72%

Productivity Improvement: 356%

Comparison of the median effort required for implemented equivalent applications using the traditional approach to software development and using the PEMSTAR Automation and Test Framework (PATF).