Published in the Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE),
San Diego, USA, November 2001, pp. forthcoming.

# Unfriendly COTS Integration – Instrumentation and Interfaces for Improved Plugability

Alexander Egyed
*Teknowledge Corporation*
*4640 Admiralty Way, Suite 231*
*Marina Del Rey, CA 90292, USA*
*aegyed@ieee.org*

Robert Balzer
*Teknowledge Corporation*
*4640 Admiralty Way, Suite 231*
*Marina Del Rey, CA 90292, USA*
*balzer@teknowledge.com*

## Abstract

*It is becoming increasingly desirable to incorporate Commercial-off-the-Shelf (COTS) tools as software components into larger software systems. Due to their large user base, COTS tools tend to be cheap, reasonably reliable, and functionally powerful. Reusing them as components has the benefit of significantly reducing development cost and effort.*

*Despite these advantages, developers encounter major obstacles in integrating most COTS tools because these tools have been constructed as stand-alone applications and make assumptions about their environment that do not hold when used as part of larger software systems. Most significantly, while they frequently contain programmatic interfaces that allow other components to obtain services from them on a direct call basis, they almost always lack the notification and data synchronicity facilities required for active integration.*

*In this paper, we present an integration framework for adding these notification and data synchronization facilities to COTS tools so that they can be integrated as active software components into larger systems. We illustrate our integration framework through tool suites we constructed around Mathworks' Matlab/Stateflow and Rational's Rose (two widely-used, large COTS tools). Our experience to date is that it is indeed possible to transform standalone COTS tools into software components.*

## 1. Introduction

Incorporating Commercial-off-the-Shelf (COTS) tools into new and existing software systems has found strong and widespread acceptance in software development. There are many advantages in doing so. As a result of their large user base, COTS tools usually have stable interfaces (APIs) and are fairly reliable. Their large user base also makes COTS tools more generic and thus functionally powerful since they often have to satisfy different user groups with different needs and goals. COTS tools also tend to represent large pieces of software, much larger than those of reusable source code libraries. Thus the ability to reuse even a single COTS tool can significantly reduce development cost and effort [3]. All these features make COTS tools very attractive reuse targets in the wake of exploding software development costs.

However, the lack of source code requires COTS tool reuse be treated differently than code reuse [1,3,15]. COTS tools cannot be tailored from "within" by modifying their code. Instead, changes must be imposed from the outside via wrappers or glue code [4,7]. Thus, incorporating COTS tools into software systems is risky [10,15]. "The fact is that using COTS software brings with it a host of unique risks quite different from those associated with software developed in-house." [3]

In an "ideal world," COTS tools would be built with complete and unrestricted access its data stores and functionality. In an "ideal world" these COTS tools could be customized to their surroundings so that they can instigate interaction like in-house developed components. Sadly, "real world" COTS tools are often only partially accessible and customizable, greatly limiting their reuse.

This paper proposes an architectural framework for tightly integrating COTS tools with other components by augmenting those COTS tools with change notifications that enable the other components to remain synchronized with the evolving data maintained by the COTS tool.

We will illustrate the use of this COTS integration framework on two complex, large-scale software systems Mathwork's Matlab/Stateflow and Rational's Rational Rose to demonstrate the tight, active integration that can be achieved between those tools and several components we developed.

While our framework will not work for all COTS integration projects, our experience in applying it to several major tools (e.g. [19]) indicates that it has broad applicability.

Section 2 presents our framework and identifies the basic interface and instrumentation technologies required to implement it. Section 3 demonstrates our framework with Matlab/Stateflow and other tools using particular interface and instrumentation technologies. Section 4 reviews the space of interface and instrumentation technologies that could be used in implementing our

framework. Sections 5, 6, and 7 describe the applicability of this work, our future plans, and conclusions.

## 2. COTS Integration Architectures

Our framework provides two types of COTS integration. The first, Directional Access, provides a standard interface for accessing and setting the data provided by a COTS tool. The second, Directional Access with Notifications, augments the first with notifications that enable interested components to track changes being made by the COTS tool to the data it provides so that they remain synchronized to the current state of that data.

### 2.1. Directional Access

The most commonly attempted (traditional) integration architecture is to have in-house-developed components access passive, service-providing COTS tools. Note that with passivity we imply that the given COTS tool has no knowledge about its surrounding components (whether they are in-house or other COTS). It is the nature of passive components not to initiate interactions with the outside world but instead to wait for service requests. From the perspective of the overall system architecture it thus appears as if integrated COTS tools are dormant unless requested to do something (although internally they may not be passive as was discussed earlier). We refer to the integration of in-house developed components with COTS tools as "directional access" since only in-house developed components can access COTS components but not vice versa.

Directional access is generally realized via wrapper or glue code [7] that forms a defined interface for a given COTS tool. The interface is then used by other software components to interact with the COTS tool (Figure 1 top). Usually, the interface provides methods to read and write data from/to the COTS tool (data access) or to trigger some form of processing (control access). Depending how the interface is realized, multiple "client" components may interact with the COTS tool. Our implementation follows this tradition and allows multiple clients to interact with the COTS tool. All of the particular COTS tool's APIs are hidden inside the framework's constructed interface for that tool.

As an example of Directional Access integration, we built a model transformation tool called UML/Analyzer [6] that abstracts class and object diagrams [5] created in Rational Rose (a CASE tool that supports modeling in the Unified Modeling Language (UML) [5]). It uses the framework's Access module to access the Rational Rose models. That module internally uses Rational Rose's native APIs (COM) to access the required data.

The passivity of directional access has the disadvantage that COTS components may undergo internal changes of which other, neighboring components may be unaware. For instance, the UML/Analyzer model, which is extracted from Rational Rose reflects the state of when it was last extracted; a severe deficiency since Rational Rose models can be modified concurrently by interacting users.

### 2.2. Directional Integration with Notification

While this integration framework could be applied to any COTS tool that maintains evolving data to be shared with other components, we have focused on COTS tools with a user driven GUI.

In an ideal world, a COTS tool (like Rational Rose) could be customized to notify other components (like UML/Analyzer) of internal changes (i.e., model changes). In such an ideal world, the COTS tool would become an active participant in the software system into which it is integrated. While many COTS tools provide data and control integration (e.g.., via import functions or tool add-ins features), it is rare for them to provide a tailorable set of notifications. Our framework provides a way to add data and control notification to COTS tools and architecture for using those notifications.

Figure 1 (bottom) illustrates our proposed framework for providing data and control integration to COTS tools. In our framework, active integration builds on top of passive integration by adding a *CallBack Manager* that handles notifications issued by the augmented COTS tool. The callback manager acts as a broker between the COTS tool and the potentially large set of software components that may be interested in the COTS tool's activities.

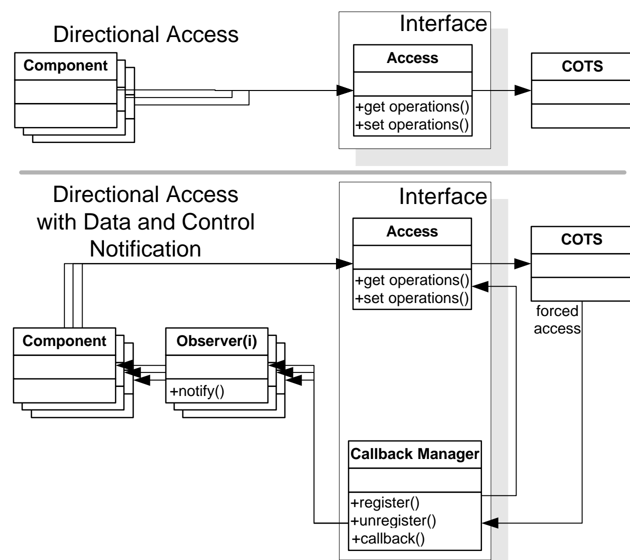


**Figure 1. (Multi-) Directional Access with/without Data and Control Notification using Observer Pattern**

Figure 1 shows the use of observers that, coupled with the callback manager, realize the interaction between components and COTS tools. Specifically, observers get registered to callback managers to indicate interest in being notified about something the COTS component has to offer (see also observer design pattern [9]). Observers are built specifically to satisfy the interaction needs of the components interested in the COTS tool's activities. The evaluation of what to notify and how to notify is left to the observer (i.e., filtering and syntactic/semantic transformations like data type conversions). If components reside on distributed nodes (e.g., different machines) the observer also has to manage transportation issues (e.g., remote method invocations [18]). Observers thus can implement "connectors" between COTS tools and other components.

Thus, Active Integration retains the ease of access to COTS tools (see direct links that connect components to the interface in Figure 1 bottom) provided by Passive Integration, but also provides notifications that allow COTS tools to initiate interactions.

In Section 3, we will demonstrate the integration of a simulator we developed with the Matlab/Stateflow. COTS tool in which the latter triggers changes to the simulator even though it is unaware of the existence of the simulator.

Importantly, the augmented access upon which the notifications rely, are encapsulated via a well-defined and sound architectural framework. The use of a sound architectural framework in turn makes it easier to plug COTS tools into larger software systems. Our framework thus improves the "plugability" of COTS tools; the lack of which is generally seen as a significant reason for failures during component-based development [3,10]. For instance, we will show later that we can independently operate our statechart simulator on either Matlab/Stateflow or Rational Rose with only minor changes to the source code of the simulator. This form of "plugability" makes it possible to replace COTS tools in software systems with only minimal impact onto other components.

Conceptually, directional access with notification realizes bi-directional integration between any given set of COTS tools and in-house developed component. The reason why we call this integration "directional" (implying one direction) is because our framework only needs to provide a standard interface for the COTS tools (via our Access interface) for in-house developed components but not vice versa since those components have public interfaces (or can be made to have public interfaces). The data and control notification mechanisms that the framework augments can then use the provided, public interfaces of other components to interact with them.

Normally, interactions between COTS tool and other components are asynchronous (i.e., in case of change notifications), however, synchronous interactions (via locking and unlocking methods) are also supported. These are discussed in the Matlab/Stateflow case study in the next section.

Directional integration with notification is quite powerful since it enables the integrator to exert a great degree of control over COTS tools. Indeed, we found that directional integration with notification was sufficient for all of our group's COTS integration needs. Specifically, we used this form of integration to augment the public interface of Microsoft Word for added security features [19], Microsoft PowerPoint for richer modeling features [11], and Rational Rose for model access.

## 2.3. Choice of Architecture

Choosing an integration architecture depends on the interaction needs of involved components. In some cases, non-functional qualities are also important decision factors. For instance, directional access with data and control notification can be used to create and maintain consistent local copies of COTS data stores to improve access speed, i.e., in cases where interacting with the COTS tool is time consuming due to marshalling and de-marshalling in COM or CORBA. Naturally, the choice of integration type also affects other qualities like reliability, robustness, or security for the same reasons.

Our integration framework is not only incrementally constructible but is also incrementally useable. Figure 1 demonstrates incremental construction in that directional access is part of directional access with notification. It is also possible to have multi-directional access with notifications between COTS tools using mirror images of several directional accesses with notifications. This discussion is outside the scope of this paper.

Our integration framework can also support multiple architectures simultaneously. In the next section we will show that the COTS tool Matlab/Stateflow can simultaneously interface with one component (the Model Browser) via directional access, with another component (the SDS Simulator [8]) via directional access with notification, and with a COTS component (Rational Rose) via multi-directional access with notification.

Our framework provides technologies for externalizing internal activities of COTS tools via hooks [2]; and our framework provides technologies for realizing architectural infrastructures that make COTS components appear like generic (in-house-developed) architectural components. Thus, directional access with notification is the COTS equivalent to multi-directional access between in-house developed components.

## 3. Matlab/Stateflow Case Study

This section illustrates the use of directional access with/without data and control notification in the context of the COTS tool Matlab/Stateflow and two in-house developed tools called *Model Browser* and *SDS Simulator[9]*. It uses particular interface and instrumentation technologies for the Case Study. The space of such technologies from which these particular instances were selected is presented in the next section.

Matlab provides a powerful modeling environment for real-time embedded systems and is widely used in the automotive and aerospace industry to simulate and validate complex problems. In support of the MoBIES project (Model-Based Integration of Embedded Systems) we were asked to provide an integration framework for COTS tools that are commonly used in that community. The lack of such integration is generally seen as a major deterrent to model-based development supported by multiple tools.

### 3.1 Directional Access

Although Matlab/Stateflow does not provide a public interface, its developers at Mathworks built an undocumented interface. For integration purpose this distinction is not significant, except for the potential lack of its stability in future versions.

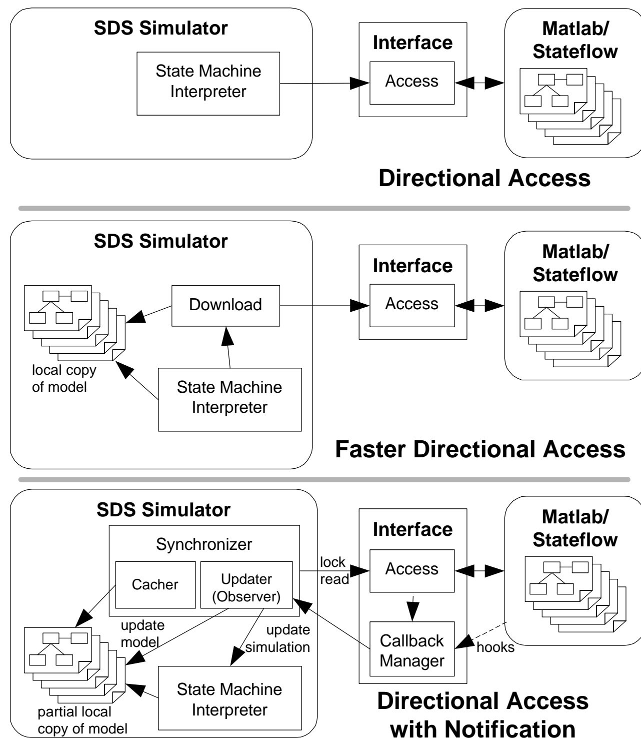


**Figure 2. Directional Access without (top,middle) and with (bottom) notification in context of Matlab/Stateflow**

To provide a more generic and stable interface for Matlab/Stateflow and to make its interface more widely accessible, we decided to build an intermediate access layer on top of Matlab/Stateflow's native APIs. This access layer was implemented in the form of directional access as depicted in Figure 1 (top). The interface implements the UML statechart meta-model and also uses the COM middleware platform for accessibility. As such, the new interface provides COM classes like *State*, *Transition*, *Event*, or *Trigger* to access Matlab/Stateflow statechart elements. Making use of the new public entry point to Matlab/Stateflow, we adapted a series of tools that we had developed to interact with it. For instance, we modified a model browser and a simulator (Figure 2 top) to browse or simulate the model described in Matlab/Stateflow. Note that we replaced Matlab/Stateflow's simulator with our simulator because we enhanced the textual notation of statecharts [8]. We thus wanted to continue using Matlab/Stateflow's user interface but have Matlab/Stateflow use our simulator instead of its built-in one (control integration).

The model browser aids developers in understanding statechart models by providing different ways of navigating those models while our simulator uses Matlab/Stateflow solely as its user interface for drawing purposesto take advantage of users familiarity with Matlab/Stateflow.

We found that these tools had quite different interaction needs with Matlab/Stateflow. For instance, the simulator's performance was initially very slow mainly due to frequent, time-consuming interactions with the COTS tool (Figure 2 top). Also, since Matlab/Stateflow maintains its own user interface we encountered complex synchronicity problems when a user changed the Matlab/Stateflow model during simulation. Although the actual interface to Matlab/Stateflow was very reliable, the chosen architectural style led to a fragile integration with serious data synchronicity problems. We encountered similar integration problems with other COTS tools like Microsoft PowerPoint and Rational Rose.

### 3.2 Faster Directional Access

Within the context of Directional Access, this performance problem could be improved by downloading a local copy of the COTS tool's data model before beginning the simulation. Figure 2 (middle) depicts this better solution and shows that the statechart simulator (SDS Simulator) consists of its own copy of the statechart model, a download component that uses Matlab/Stateflow's provided interface to create the local copy, and an interpreter component that is doing the actual simulation. This solution is an improvement in terms of performance because accessing the model via the

cached local copy (once established) is faster than inter-process COM calls to Matlab/Stateflow.

This solution also improves, but does not eliminate, the model synchronicity problems. The new solution is only able to simulate the latest downloaded version of the model but not its current state (i.e., it would be beneficial if a developer could change the model during simulation to fix or simulate defects). Also, the new solution may still encounter synchronicity problems if, during download, the Matlab/Stateflow model is being changed.

### 3.3 Directional Access with Notification

While the faster Directional access was satisfactory in providing a high performance integration with the SDS Simulator, the remaining reliability problems are architectural in nature and cannot be improved with traditional COTS integration frameworks (i.e. by Directional Access alone). To provide better integration between the simulator and the COTS tool, three major challenges have to be resolved:

- Prevent users from making changes to Matlab/Stateflow while download is in progress
- Update the local copy of the simulator whenever the Matlab/Stateflow model changes
- Update the current simulation whenever the simulation is active (running) and the Matlab/Stateflow model changes

Architecturally, to resolve all these challenges Matlab/Stateflow must become an active component in interacting with its neighbors. Directional access with notification provides an architectural framework for doing exactly that (Figure 2 bottom).

What we need to accomplish is a means of getting a notification from Matlab/Stateflow whenever the user performs an action that changes Matlab/Stateflow's model. Via hooks [2], we can monitor operating system interactions with Matlab/Stateflow. In particular, user inputs such as key strokes or mouse clicks can be monitored. It requires very little code (roughly 100 lines of C code, see pseudo code of hooks() in Figure 3) to build hooks for Matlab/Stateflow to intercept keyboard and mouse events and to invoke the Callback Function in the Callback Manager when these events occur(see also Figure 2 bottom and Figure 3).

This callback function is responsible for translating the low-level events it receives from the hooks that instruments the COTS tool into high-level events of interest to the components being integrated with the COTS tool. In particular, it determines whether any components of the model being shared with these components have been changed by the COTS tool. To make this determination, it caches a copy of this model and compares the model within the COTS tool with this cached copy. It optimizes this comparison by limiting it to only the set of model elements currently selected (because Matlab/Stateflow, like many other COTS tools such as PowerPoint and Rational Rose, limit user changes to only those elements of the model that are selected in their GUI).

Our architecture separates this COTS tool specific filtering, which translates low-level events into high-level integration notifications, from the hooks that instrument the COTS tool's operation. The callback manager encapsulates this filtering and provides registration services for the notifications produced by this filtering.

The callback manager is packaged with the access methods within a single COM interface to simplify their use by both the external software components wishing to interact with a given COTS tool and by the instrumented code inside the COTS tool providing low-level notifications of relevant user activity.

The observers responsibility is to actually realize how callbacks to interested components are performed; i.e., possibly filtering unimportant messages and performing syntactic and semantic transformations. In Figure 3 the *Updater* observer is a COM object and COM handles the notification. The *Updater* itself performs model

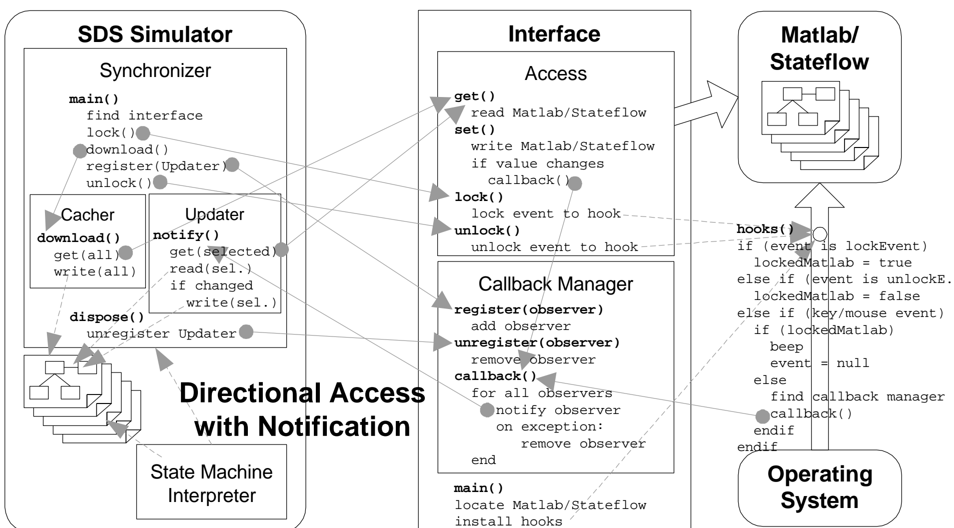


**Figure 3. Pseudocode of Matlab/Stateflow and SDS Simulator Integration (this figure is a refinement of Figure 2 bottom).**

synchronization within the simulator once it is notified by Callback Manager that the COTS tool's model has changed.

Observers are not automatically created as part of a component's interface to a COTS tool. Instead, only a "template" is provided for constructing such observers so that their interfaces conform to our frameworks architecture for interfacing with the callback manager. In essence, the observer only needs to provide one method called *notify* that receives notifications from the CallBack Manager.

In case of the SDS Simulator, updating the local copy is only one of two synchronization tasks that its observer must perform. The second task is to update the running simulation if it is currently active (not depicted in Figure 3 for brevity). For instance, guard conditions might be updated as the simulation unfolds. Changes could also occur that invalidate the simulation (i.e., the current state is deleted). In such a case, the observer can shut down the current simulation or proceed with a defined recovery process.

The above discussion demonstrates how directional access with notification can be used to solve both the data synchronization problem (data integration) and the simulator update problem (control integration).

Our SDS Simulator also needed the ability to prevent the shared COTS tool's model from changing during certain critical periods. This capability was achieved by using the hooks to actively block user inputs to the COTS tool during these critical periods and to issue a "beep": to let the user know that their input was temporarily blocked. To notify our hooks to lock or unlock Matlab/Stateflow's GUI, our interface sends events to Matlab/Stateflow that are intercepted by our hooks (see Figure 3).

# 4. Interface & Instrumentation Technologies

Our case study relied on COM and hooks to interface and instrument Matlab/Stateflow. This section compares these and additional technologies.

## 4.1. Provided Access

Developers are dependent on public interfaces to interact with COTS tools (data and control). For data access the most commonly found public interface is the persistent data storage (i.e., files or databases). Although, persistent data formats may be undocumented, it is often not very hard to create parsers that can read them. Accessing persistent data, however, only provides data access to COTS tools and, even then, only access to data that reflects the COTS tool during start-up or last saving. For more interactive access to COTS tools, middleware platforms (e.g., COM [22], CORBA [16,21], DLL, RMI [18]) can be used. Although, middleware platforms are capable of supporting both run-time data and run-time control access to COTS tools, most COTS tools do not natively provide comprehensive interfaces for such access.

**Table 1. Provided Access to COTS Tools**

| ***Method*** | ***Description*** |
|---|---|
| *Persistent Data Access* | Parsing data files or databases. Works well in cases where the file is kept consistent with the internal model of the running COTS tool. |
| *Middleware Access* | Run-time access via middleware platforms like COM, CORBA, or RMI. Works well in cases where COTS developers foresaw intended usages and provided necessary access points. |

## 4.2. Augmented Access

Because the natively provided interfaces to COTS tools is often so restricted, a variety of techniques have been developed to augment this natively provided access.

One general augmented access technique is *Hooking* which attaches code to the external or internal interfaces of a component (Balzer and Goldman [2]). These Hooks may be purely passive – only observing COTS tool behavior. But they may also be active – generating new operations, modifying existing ones, or blocking them.

Another form of augmented access to a COTS tool is via its binary representation. Given enough understanding of a COTS tool's binary code, the code could be "patched" to replace, delete, or add new functionalities. This form of COTS access, however, requires low-level familiarity with the machine code of COTS components.

While these augmented access techniques require tool-specific development and may be obsoleted by new releases, they nonetheless provide access to otherwise inaccessible data and control.

## 4.3. Hybrid Access (Provided + Augmented)

A common example where a combination of provided and augmented access is important is a COTS tool that provides access to its internal state but does not provide change notification.

**Table 2. Augmented Access to COTS Tools**

| ***Method*** | ***Description*** |
|---|---|
| *Hooks* | Intercept communications to and from COTS components (or between sub components of the same COTS tools). Can be used to passively observe or actively manipulate COTS tools. |
| *Memory / Executable Patching* | Modifying the binary representation of COTS tools either on the persistent data level or during run-time in the RAM. Although very operating system dependent, low-level alterations to the memory space of the COTS tool can be used to analyze and manipulate. |

Our case study discussed earlier contains such an example. Matlab/Stateflow allows statechart diagrams to be created and modified. Its provided (public) interface provides access to its model. However, after this data has been accessed, a Matlab/Stateflow user may continue to alter the model via the tool's user interface, making the previously read data inconsistent with the current state of the COTS tool.

By combining its provided interfaces with augmented access a data synchronized interface was created. This combined use of public and augmented access methods has been applied very effectively to several other COTS tools including Rational Rose, Microsoft PowerPoint, and Microsoft Word.

## 5. Discussion

### 5.1 Clean Interfaces for COTS Tool Integration

The Case Study illustrated the integration of a specific COTS tool with newly developed components using our integration framework. Our framework provided a standardized interface for those components to interact with the COTS tool for both access and notifications. Those components were architecturally separated from the low-level mechanisms required to support the COTS tool's side of this interface.

While the detailed issues of interfacing to, and augmenting, the native capabilities of COTS tools have not been eliminated, our framework provides a standardized way of encapsulating them and cleanly interacting with them through generic interfaces.

This is an important separation since a COTS tool, once instrumented and encapsulated with an adequate interface, becomes a true architectural component that can be integrated with a wide variety of other components. Our framework thus shows that separation of concerns [20] is possible even when COTS tools are being integrated with other components.

As an example of this separation, our simulator can use either Rational Rose or Matlab/Stateflow as its graphical front-end with only minor changes to its code (Figure 4).

Note that the generic usefulness of a COTS component still depends on the quality of the implemented interface. Nonetheless, the interface can be augmented later if there is a need. It is also important to note that neither the hooks, nor the concept of change notifications are new. They have existed for some time (i.e,. message passing in development environments [17]) . The accomplishment of this work is our framework for integrating complex, large-scale, and partially-accessible COTS tools through standardized interfaces that encapsulate access and change notification augmentations of their native capabilities.

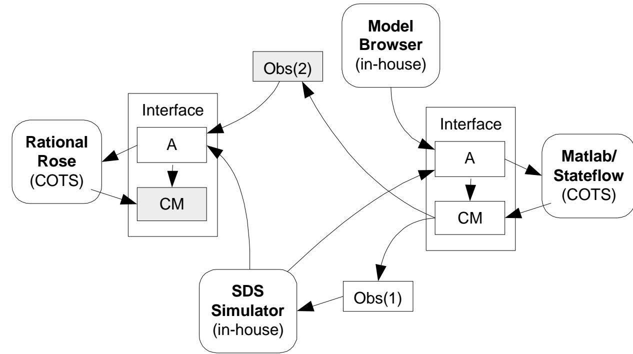


**Figure 4. Multi-Dimensional Integration with Matlab/Stateflow**

While the example integrations presented in this paper employ a narrow set of implementation technologies, our framework is generic and can employ the entire set of technologies discussed in the Interface and Instrumentation section.

Our integration framework is generic in the set of architectural styles it supports. For instance, architecture description languages [14] (ADLs) often use distinct interaction technologies and protocols. As such, components may use synchronous calls (i.e., Main-Subroutine Style), asynchronous calls (i.e., RMI), events [12], shared memory, explicit data connectors [13], middleware platforms (i.e., COM or CORBA) or other communication methods. This abundance of interaction methods implies many different architectural styles.

Our framework treats a COTS tool with its access methods and callback manager as a single software component. The choice of architectural style determines how the interactions between this (enhanced) COTS component and others will be realized. We thus do not build observers for these COTS components but implement them as part of the components that need to interact with them.

### 5.2 Improved Plugability

Thus far, we have discussed integration styles, topologies, and infrastructures separately. They can nevertheless be mixed within a single system. Figure 4 depicts the integration of multiple COTS tools (Rational Rose and Matlab/Stateflow) and components we developed (Model Browser and SDS Simulator) using our integration framework (Rose change notification and observers are not fully implemented and thus shown in gray). As can be seen, each of our components uses a different framework of integrating with Matlab/Stateflow:

1. the Model Browser uses directional access (Figure 1 top);
2. the SDS Simulator uses directional access with data and control notification to talk to Matlab/Stateflow

(Figure 1 bottom) and simple directional access (Figure 1 top) to talk to Rational Rose;

3. the Rational Rose COTS tool uses uni-directional access with data notification (Figure 1 bottom) to talk to Matlab/Stateflow.

Our case study thus demonstrates that integration frameworks can be mixed during run-time to satisfy individual integration needs.

Our framework improves the plugability of COTS tools because newly developed components, like the SDS Simulator, can be built under the assumption that it is being integrated with an idealized COTS component. The simulator can thus be made insensitive to the particular choice of COTS tool (Rational Rose, Matlab/Stateflow, or some other tool) being used as its graphical front-end.

While we take the stance that reuse is significantly better than re-development, it may not always be possible to reuse a particular COTS tool given the limitations of augmented access. There is a non-obvious trade-off between the cost of re-development and reuse, and given the diverse nature of COTS tools there is no simple way of predicting which is better.

### 5.3 Open Issues

While our integration framework provides standardized interfaces that encapsulate access and change notification augmentations of COTS tools' native capabilities, it does not dictate how those interfaces should be realized. This keeps our solution generic and lightweight.

Also, our work does not address the versioning problem that is inevitable with COTS tools. A new version of a COTS tool may not be compatible with previous augmentations occurring within our encapsulated interface. While we have experienced only minor incompatibilities that were easily resolved with the COTS tools we have been working with, it is certainly possible that major incompatibilities could occur. This possibility would make it more difficult and resource consuming to upgrade those COTS tools.

Finally, while our integration framework could be applied to any COTS tool that maintains evolving data to be shared with other components, we have focused on COTS tools with a user driven GUI. This has limited our experience with the broader set of COTS tools that might be integrated with other components.

## 6. Conclusion

Reusing commercial-off-the-shelf tools (COTS) has the potential of significantly improving software development speed and cost. However, COTS reuse also introduces new complexities we have traditionally been ill equipped to handle. This paper discussed how augmented access to COTS tools can complement their native access to increase the observeability and usability of those COTS tools. It also showed how this technology can be used to interface and instrument COTS tools so that they have a clean, standardized interface that can be "plugged" into a wide variety of software systems. We discussed two basic integration frameworks that can be used concurrently and interchangeably and we discussed different integration topologies and styles. The paper thus contributes a framework for adding data and control notification to traditional directional access methods.

Although, we do not claim that all COTS tools can be integrated via our framework, we do believe that it has wide applicability. We demonstrated our framework in validating it on several large-scale, commercial tools such as Matlab/Stateflow, Microsoft PowerPoint, Microsoft Word, and Rational Rose as well as several tools we developed like the Model Browser, the SDS Simulator, and UML/Analyzer. We will continue to validate our approach on other COTS tools.

## Acknowledgements

Our thanks to Neil Goldman, Marcelo Tallis, Dave Wile, and all anonymous reviewers. This work was supported by DARPA under agreements F30602-00-C-0218, F30602-99-1-0524, and F30602-00-C-0200.

## References

[1] C. Abts and B.. Boehm (editors). *Proceedings of the Focused Workshop on System Integration with Commercial-Off-The-Shelf (COTS) Software*, Los Angeles: University of Southern California (USC), 1996.

[2] Balzer, R. and Goldman, N., "Mediating Connectors," *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems,* pp. 73-77, May 1999.

[3] B.W. Boehm, C. Abts, A.W. Brown, W. Chulani, B.K. Clark, E. Horowitz, R. Madacy, D. Reifer, and B. Steece. *Software Cost Estimation with COCOMO II*, New Jersey: Prentice Hall, 2000.

[4] Boehm, B. and Abts, C., COTS Integration: Plug and Pray? *IEEE Computer*, vol. 32, pp. 135-138, 1999.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[6] Egyed, A., "Semantic Abstraction Rules for Class Diagrams," *Proceedings of the 15th IEEE*

*International Conference of Automated Software Engineering (ASE),* Sept. 2000.

[7] Egyed, A., Medvidovic, N., and Gacek, C., A Component-Based Perspective on Software Mismatch Detection and Resolution *IEE Proceedings Software*, vol. 147, pp. 225-236, 2000.

[8] Egyed, A. and Wile, D., "Statechart Simulator for Modeling Architectural Dynamics," *Proceedings of the 2nd Working International Conference on Software Architecture (WICSA),* Amsterdam, The Netherlands, pp. 87-96, Aug. 2001.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reuseable Object-Oriented Software*, Addison Wesley, 1994.

[10] Garlan, D., Allen, R., and Ockerbloom, J., Architectural Mismatch or Why it's hard to build systems out of existing parts *IEEE Software*, vol. pp. 17-26, Nov, 1995.

[11] Goldman, N. and Balzer, R., "The ISI Visual Editor Generator," *Proceedings of the IEEE Symposium on Visual Languages,* 1998.

[12] Luckham, D. C. and J. Vera, J., An Event-Based Architecture Definition Language *IEEE Transactions on Software Engineering*, vol. Sep, 1995.

[13] Medvidovic, N., Rosenblum, D. S., and Taylor, R. N., "A Language and Environment for Architecture-Based Software Development and Evolution," *Proceedings of the 21st International Conference on Software Engineering (ICSE'99),* pp. 44-53, May 1999.

[14] Medvidovic, N. and Taylor, R. N., A Classification and Comparison Framework for Software Architecture Description Languages *IEEE Transactions on Software Engineering*, vol. 26, pp. 70-93, Jan, 2000.

[15] Morisio, M. , Seaman, C. B., Parra, A. T., Basili, V. R., Kraft, S. E., and Condon, S. E., "Investigating and Improving a COTS-Based Development Process," *Proceedings of the 22nd International Conference on Software Engineering (ICSE),* pp. 32-41, June 2000.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1995.

[17] Reiss, S. P., Connecting Tools Using Message Passing in the Field *IEEE Software*, vol. 7, pp. 57-66, Jul, 1990.

[18] Sun Microsystems. *Java Remote Method Invocation - Distributed Computing for Java*, 2001. (UnPub)

[19] Tallis, M. and Balzer, R., "Document Integrity through Mediated Interfaces," *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX),* 2001.

[20] Tarr, P., Osher, H., Harrison, W., and Sutton, S. M. Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proceedings of the 21st International Conference on Software Engineering (ICSE 21),* Los Angeles, CA, pp. 107-119, May 1999.

[21] Vinoski, S. , CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments *IEEE Communications Magazine*, vol. Feb, 1997.

[22] Williams, S. and Kindel, C., The Component Object Model: A Technical Overview *Dr. Dobb's Journal*, vol. Dec, 1994.