Proceedings of the 4th International Workshop on Adoption-Centric Software Engineering, co-located with ICSE 2004, Edinburgh, Scotland, UK, May 2004.

Data and State Synchronicity Problems While Integrating COTS Software into Systems

Alexander Egyed, Sven Johann, and Robert Balzer
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
{aegyed,balzer}@teknowledge.com

Abstract

The cooperation between commercial-off-the-shelf (COTS) software and in-house software within larger software systems is becoming increasingly desirable. Unfortunately, COTS software packages typically are standalone applications that do not provide information about changes to their data and/or state to the rest of the system. Furthermore, they typically use their own, proprietary data formats, which are syntactically and semantically different from the system.

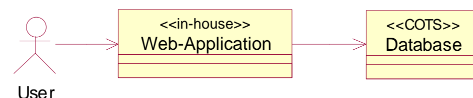
We developed an approach that externalizes data and state changes of COTS software instantly and incrementally. The approach uses a combination of instrumentation and reasoning to determine when and where changes happen inside the COTS software. It then notifies interested third parties of these changes. The approach is most useful in domains where the system has to be aware of data and state changes within COTS software but it is computationally infeasible to poll it periodically (i.e., large-scale data). The approach has been validated on several COTS design tools with large, industrial models (e.g., over 34,000 model elements) for effectiveness and scalability.

1. Introduction

Incorporating COTS software into software systems is a desirable albeit difficult challenge. It is desirable because COTS software typically represents large, reliable software that is inexpensive to buy. It is challenging because software integrators have to live with an almost complete lack of control over the COTS software products [2].

We define a COTS-based system to be a software system that includes COTS software [3]. From a software architecture perspective, a software system consists of a set of interacting software components. We thus also refer to COTS software used within a COTS-based system as COTS components. A COTS-based system may include one or more COTS components among the set of its software components.

Incorporating COTS software into new and existing COTS-based systems has found strong and widespread acceptance in software development [1]. For example, a very common integration case is in building web-based technologies on well-understood and accepted COTS web servers. Indeed, COTS integration is so well accepted in this domain that virtually no web designer would consider building a web server or a database anew.



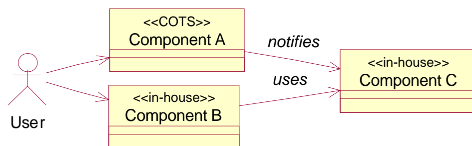
While there are many success stories that point to the seaming ease of COTS integration there are also many failures. We believe that many failures are the result of not understanding the role the COTS software is supposed to play in the software system. In other words, the main reason of failure is architectural.

Many software systems, such as the web application above, use the COTS software as a back-end; its use is primarily that of a service-providing component. Only some form of programmatic interface (API) is required for COTS software to support this use. COTS software vendors have become increasingly good at providing programmatic interfaces to data and services of their products.

COTS integration is less trivial if the COTS software becomes (part of) the front end; e.g., with its native user interface exposed and available to the user. These cases are rather complex because the COTS software may undergo user-induced changes (through its native user

interface) that are not readily observable through the programmatic interface. In other words, the challenge of COTS integration is in maintaining the state (e.g. data model, configuration, etc) of the COTS software consistent with the overall state of the system even while users manipulate the system through the COTS native user interface.

This paper discusses this issue from the perspective of using COTS design tools such as IBM Rational Rose, Mathwork's Matlab, or Microsoft PowerPoint. These design tools are well-accepted COTS software products. They exhibit commonly understood, graphical user interfaces. This paper shows how to use these COTS products, with their accepted user interfaces, to build a functioning software system where the architectural integration problem is more like this:



2. Data Consistency and State Synchronicity

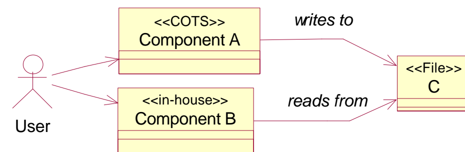
Most COTS software products initiate interaction with other software products they are explicitly designed to interact with. This is problematic because there are integration scenarios where COTS software is required to interact with software it was not explicitly designed to interact with. Moreover, while it is true in some cases that COTS software is fully proactive (with respect to changes to its internal state and data), we found that COTS software with user-driven GUIs (graphical user interface) tend to be less proactive. This raises the severe problem of maintaining the consistency and synchronicity of shared data and state between such a COTS component and its system while a user is manipulating it. The challenges are:

1) Data Inconsistency: Data captured in COTS software may be consumed by other components in a system. If a user manipulates the data within the COTS software then this may introduce inconsistency in the shared data. The problem is that COTS software typically does not know or care about notifying other components of internal changes.
2) State Synchronicity: User actions in COTS software may have system relevance in some cases. COTS software does not understand the needs of a system it is part of and consequently does not recognize user actions the system must be notified about. System relevant user actions may get lost if they are done through the native user interface of the COTS software.

In an ideal world, COTS software is configurable to notify other components of relevant internal changes (data and state). In such an ideal world, the COTS software becomes an active participant in the COTS-based system into which it is being integrated. Today it is rare for COTS software to have these capabilities built-in. For integration, this creates a major challenge of how such COTS software can be augmented from the "outside" so that internal activities (state and data changes) relevant to other components are proactively communicated to them. The next sections discusses how this can be accomplished using a combination of instrumentation and reasoning.

3. Batch Change Detection

Batch notification is to externalize all relevant COTS state information (e.g., by exporting design data from Rational Rose) so that it becomes available to other components. This is probably the easiest solution but has one major drawback in that it is a loose form of incorporating COTS software into software systems. State information has to be replicated with the drawback that even minor state changes to the COTS software cause major synchronization activities (e.g., complete re-export of all design data from Rational Rose).



Batch notification is computationally expensive and only then feasible if the integration between the COTS software and the rest of the system is loose or the amount of data/state information is small.

The following will present our approach to incremental notification. Our approach wraps COTS software to identify changes as they occur with several beneficial side effects:

- Only changes are forwarded
- Changes are forwarded instantly
- Change detection is fully automated

4. Incremental Change Detection

To understand COTS changes one has to be aware of *when* and *where* the COTS software undergoes changes. This problem does not exist with in-house developed components because they can be programmed to notify other, affected components of a system. However, the lack of access to the source code makes this approach impossible for COTS software. Therefore, to incorporate

COTS software into systems the COTS software must be observed to identify and forward all relevant events to the system.

4.1. The *When* and *Where* of Change Detection

There is a trivial albeit computationally infeasible approach to observing changes in COTS software by caching its state and continuously comparing its current state with the cached state. Unfortunately, this approach does not scale well if the COTS state consists of large amounts of data (e.g., design data in tools such as IBM Rational Rose, Mathwork's Matlab, or Microsoft PowerPoint). To better observe changes in COTS software it has to be understood *where* and *when* changes happen.

Take, for example, IBM Rational Rose. In Rose a user creates a new class element by clicking on a toolbar button ("new class") followed by clicking on some free space in the adjacent class diagram. A class icon appears on the diagram and the icon is initially marked as selected. By then clicking on the newly created, selected class icon once more, the name of the class may be changed from its default or class features such as methods and attributes may be added. These changes could also be done by double-clicking on the class icon to open a specification window. There are two patterns worth observing at this point:

- Changes happen in response to mouse and keyboard events only
- Changes happen to selected elements only

The first observation is critical in telling *when* changes happen. It is not necessary to perform (potentially computationally expensive) change detection while no user activity is observed. The second observation is critical in telling *where* changes happen (Egyed and Balzer 2001). It is not necessary to perform (potentially computationally expensive) change detection on the entire COTS design data (state) but only on the limited data that is selected at any given time.

Both observations are the key for scalable and reliable change detection in GUI-driven COTS software.

4.2. Caching and Comparison

Changes are detected by comparing a previous state of a COTS software with its current state. Generally, this implies a comparison of the previously cached state with the current one. Knowing the time when changes happen and the location where changes happen limits what and when to compare.

Basic Change Detection

After every mouse/keyboard event, we ask Rose what elements are selected (via its programmatic interface) to compare these elements with the ones we cached previously. If we find a difference (e.g. a changed name, a new method) between the cached elements and the selected ones then we notify other components (e.g., in-house developed components) about this difference. Thus, our approach notifies other components on the behalf of the COTS software. If we find a difference then we also update the cache to ensure that differences are reported once only. Obviously, the effort of finding changes is computationally cheap because a user tends to work with few design elements at any given time only.

Our approach uses the programmatic interface of the COTS software to elicit and cache state information. For example, in IBM Rational Rose this includes design data such as classes, relationships, etc. The caching is limited to "relevant" state information that is of interest to other components of the system. For example, if it is desired to integrate some class diagram analysis tool with Rose, then change detection may be limited to class diagrams only (i.e., ignoring sequence diagrams, state chart diagrams). In summary, basic change detection is as follows:

1. download and cache data when first selected
2. re-download data and compare with cached data when de-selected
3. update cache
4. no need to cache selected element a second time if it was selected previously because the cache stays up-to-date

This approach detects changes between the cached and the current state. However, there are two special cases: 1) new elements cannot be compared because they have never been cached and 2) deleted elements cannot be compared because they do not exist in the COTS software any more.

The creation and deletion problem can be addressed as follows. If we cannot find a cached element for a selected one then this implies that it was newly created (otherwise it would have been cached earlier[1]). We then notify other components of the newly created element and create a cached element for future comparison. In reverse, if an element in the cache does not exist in the COTS software then it was deleted. Other components are thus notified of this deletion and the cached element is deleted as well. Note that a deletion is only detected after de-selection (i.e., a deleted element is a previously selected element

[1] Note: we pre-cache all data identifications initially to understand the difference between newly created data and old data that was never selected.

that was deleted) and a creation can only be detected after selection.

Ripple Effect of Change Detection

Until now, we claimed that changes happen to selected elements only. This is not always correct. Certain changes to selected elements may trigger changes to "adjacent", non-selected elements. For example, if a class X has a relationship to class Y then the deletion of class X also causes the deletion of the relationship between X and Y although the latter is never selected.

There are two ways of handling this ripple effect. The easiest way is to redefine selection to include all elements that are affected by a change. For example, if a class is selected then we define that all its relationships are selected also. Change detection then compares the class and its relationships. This approach works well if the ripple effect does not affect many adjacent elements (e.g., as in this example) but it could become computational expensive.

A harder but more efficient way of handling the ripple effect is to implement how changes in selected elements affect other, non-selected elements. For example, we implemented the knowledge that the deletion of a class requires its relationships to be deleted also. In this case, neither the creation of a class nor its change does have the same ripple effect.

Anomalies

We found that basic change detection and their ripple effects cover most scenarios for change detection. However, there are exceptions that cannot be handled in a disciplined manner. We found only few scenarios in Rose that had to be handled differently.

For example, the state machines in Rose have a peculiar bug in that it is possible to drag-and-drop them into different classes while the programmatic interface to Rose does not realize this. If, in the current version of Rose, a state machine is moved from class A to class B then, strangely, both classes A and B believe they own the state machine although only one of them does. We thus had to tweak our approach to also consider the qualified name of a state machine (a hierarchical identifier) to identify the correct response from Rose. Obviously, this solution is very specific to this anomaly. Fortunately, not many such anomalies exist.

5. Change Detection Infrastructure

Figure 1 depicts our infrastructure for augmenting COTS software schematically [5]. The center of the figure holds the actual COTS software. Since no source code is available, it cannot be changed from within. *Instrumentation* [6] is used to monitor outside stimuli directed towards the COTS software (shaded frame around the COTS software). For example, we use instrumented wrapper technology to observe interactions between a software component and its environment (e.g., mouse and keyboard events). A customized *Reasoning* component within the framework then uses information made available through instrumentation and from inspection of the COTS component's state and data (via its programmatic interface) to infer what internal changes this activity caused. It essentially implements Caching and Comparison discussed above.

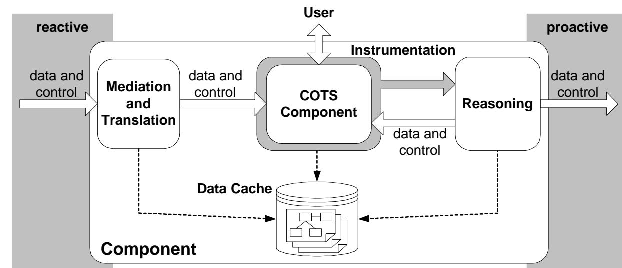


Figure 1. Augmenting a COTS Component from the Outside through Mediation, Translation, Instrumentation, and Reasoning [5]

Mediation and translation is used to construct alternative interfaces for COTS products to facilitate their use as components in larger systems. Mediators and translators [6] augment native interfaces of COTS software (i.e., wrappers or glue code). The purpose of translation is to make COTS-specific data and control information available in a format that is understood by other components of the COTS-based system (e.g., to impose a standardized interface on a COTS software). The purpose of mediation is to bridge middleware platforms (e.g., COM [15], CORBA [9,13], DLL, RMI [11]) between COTS software and other components of the system. Other components of that system then do not use the COTS native interface directly but instead use the new, augmented interface. Using an augmented interface has the advantage that the appearance of COTS software is "altered" without changing the COTS software itself (see also Figure 1). The services of the COTS software are then provided in the alternative format without other components and the COTS software being aware of this.

The optional data store is required to save cached data and other information.

6. Validation

The integration framework was validated on three major COTS products of different vendors (IBM's Rational Rose, Mathwork's Matlab/Stateflow, and Microsoft PowerPoint). Each COTS product was consequently integrated with different in-house and third-

party systems. In total, over ten integration case studies were performed that tested the validity of our approach. For example, Rose was integrated with the UML/Analyzer system for automated consistency checking between UML class diagrams and C2 architectural descriptions [12]; it was integrated with an automated class diagrams abstraction software [4], the SDS Simulator for executing UML-like class and statechart diagrams [7], the Boeing/MoBIES Translator and Exporter for modeling embedded systems [10], and several other systems. Similarly, Matlab/Stateflow and Microsoft PowerPoint were integrated into yet other systems like the Design Editor for modeling user-definable notations [8] or the survey authoring system [14].

Scalability is key to our approach. The largest model our approach was tested on was an Avionics design model from Boeing with over 43,000 model elements. While the initial pre-caching takes about 20 seconds, the subsequent caching and comparison done with every mouse or keyboard event is unnoticeable to human users. Boeing engineers and other groups have used our change detection approach without scalability issues.

Although our case studies demonstrated a wide range of applicability of our integration infrastructure, it cannot be considered proof of its general applicability. To date, our focus was primarily on COTS software with graphical user interfaces that do externalize significant parts of their internal data. In the context of these systems, we have repeatedly demonstrated that it is possible to integrate COTS software in a scalable and reliable fashion. The quality of the COTS-based systems was evaluated through numerous scalability and usability tests. To date, our infrastructure has been used by several companies (e.g., Boeing, Honeywell, and SoHaR) and universities (e.g., Carnegie Mellon University, University of Southern California, Western Michigan University).

7. Approximation

It is generally easier to maintain consistency between COTS software and the system it is being integrated with if the semantics of the COTS data is similar to the semantics of the system data. For example, we integrated Rational Rose design information with UML-compatible design information and both are conceptually similar. Unfortunately, consistency becomes more complicated if the data of COTS software is re-interpreted into a semantically different domain. This is not uncommon. For example, many applications exist that use Rose as a drawing tool. In those cases, the meaning of boxes and arrows may differ widely.

This section discusses how to "relax" change detection depending on the difficulty of the integration problem. This problem was motivated by our need to having a domain-specific component model, called the ESCM (Embedded Systems Component Model) [10], integrated with Rational Rose. While it is out of the scope to discuss the ESCM, it must be noted that its elements do not readily map one-to-one to Rose elements. As such, there are cases where the creation of an element in Rose may cause deletions in ESCM and there are cases where overlapping structures in Rose may relate to individual ESCM elements. This integration scenario is problematic because it is very elaborate to define how changes in Rose affect the ESCM.

Previously, we solved the integration problem by comparing Rose data with cached data. User actions, such as mouse and keyboard events, triggered partial re-transformations to compare the current Rose state with the cached copy. The comparison itself was trivial; so was update. The key was transformation.

The main difficulty of integrating the ESCM is in determining what to re-transform and what to compare. This is a scoping problem and it becomes more severe the more complex the relationship between system data (e.g., ESCM) and COTS data becomes. While we implemented a very precise, incremental change notification mechanism for Rose->ESCM (its discussion is out of the scope), we found that it is often good enough to approximate change detection.

Thus, we simplified the problem by implementing change detecting with the possibility of reporting false positives (Rose change is reported that does not change the ESCM) but the guarantee of not omitting true positives (Rose change that changes the ESCM). In case of integrating ESCM with Rose, it was not problematic to err on the side of reporting changes that actually did not happen since they only led to some unnecessary but harmless synchronization tasks. The ability to relax the quality of change detection (i.e., false positives) strongly improved computational complexity in this case.

8. Conclusion

Consistency between commercial-off-the-shelf software (COTS), their wrappers, and other components is a pre-condition for many COTS-based systems. Our experience is that it is possible to observe data and state changes in GUI-driven COTS software even if the COTS software vendor did not provide a (complete) programmatic interface for doing so. This paper discussed several strategies for adding change detection mechanisms to COTS software.

Acknowledgement

We wish to acknowledge Neil Goldman, Marcelo Tallis, and David Wile for their feedback and comments.

References

[1] Boehm, B., Port, D., Yang, Y., Bhuta, J., and Abts, C. Composable Process Elements for Developing COTS-Based Applications. 2002.

[2] Boehm, B.W., Abts, C., Brown, A. W., et al: Software Cost Estimation with COCOMO II. New Jersey, Prentice Hall, 2000.

[3] Brownsword L. , Oberndorf P., and Sledge C.: Developing New Processes for COTS-Based Systems. *IEEE Software*, 2000, 48-55.

[4] Egyed A.: Automated Abstraction of Class Diagrams. *ACM Transaction on Software Engineering and Methodology (TOSEM)* 11(4), 2002, 449-491.

[5] Egyed, A. and Balzer, R.: "Unfriendly COTS Integration - Instrumentation and Interfaces for Improved Plugability," *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, USA, November 2001.

[6] Egyed A., Medvidovic N., and Gacek C.: A Component-Based Perspective on Software Mismatch Detection and Resolution. *IEE Proceedings Software* 147(6), 2000, 225-236.

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

[8] Goldman, N. and Balzer, R.: "The ISI Visual Editor Generator," *Proceedings of the IEEE Symposium on Visual Languages*, Tokyo, Japan, 1999, pp.20-27.

[9] Object Management Group: The Common Object Request Broker: Architecture and Specification. 1995.

[10] Schulte, Mark. MoBIES Application Component Library Interface for the Model-Based Integration of Embedded Software Weapon System Open Experimental Platform. 2002.

[11] Sun Microsystems: Java Remote Method Invocation - Distributed Computing for Java. 2001.(UnPub)

[12] Taylor R. N., Medvidovic N., Anderson K. N., Whitehead E. J. Jr., Robbins J. E., Nies K. A., Oreizy P., and Dubrow D. L.: A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* 22(6), 1996, 390-406.

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

[14] Wile D.: Supporting the DSL Spectrum. Journal of Computing and Information Technology. *Journal on Computing and Information Technology* 9(4), 2001, 263-287.

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