State Consistency Strategies for COTS Integration

Sven Johann
University of Applied Sciences Mannheim
Windeckstrasse 110
68163 Mannheim, Germany
sven_johann@gmx.net

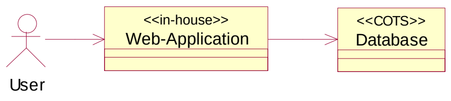
Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
aegyed@ieee.org

Abstract. The cooperation between commercial-off-the-shelf (COTS) software and in-house software within larger software systems is becoming increasingly desirable. Because COTS software is typically integrated into standalone applications, they do not provide subscription mechanisms that inform other components about internal changes. This, in combination with semantic differences in how COTS data is perceived within the integrated system, causes consistency problems. This paper will first present a scalable consistency approach for the COTS software IBM Rational Rose. The paper will then discuss this approach in context of a set of strategies for COTS integration that is heavily based on incremental transformation.

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 product.

To date, COTS software integration is not uncommon. It has become common practice to build software systems on top of COTS software. For example, a very common integration case is in building web-based technologies on well-understood and accepted web server COTS software. 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.

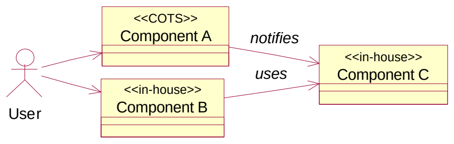


While there are many success stories that point to the seaming ease of COTS integration there are also many failures. We believe that many failures are the result of not understanding the role COTS software is supposed to play in the software system. In other words, the main reason of failure is architectural.

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 has become increasingly good at providing programmatic interfaces to data and services they provide.

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 tools; they exhibit commonly understood graphical user interfaces. This paper will show how to use these COTS tools, with their accepted user interfaces, to build a functioning software system where the architectural integration problem is more like this:



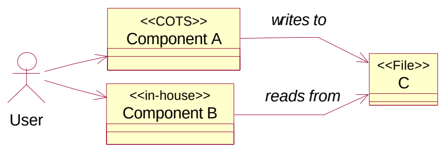
The COTS integration challenge is to make software systems aware of when and how incorporated COTS software undergoes changes. This problem does not exist with in-house developed components because they can be re-programmed to notify other, affected parts of a system (e.g., other components). The lack of access to the source code of COTS software makes this approach impossible.

Because COTS software is not originally designed to be components of a new system, they offer at most interfaces to extend their functionality with plug-ins. Their purpose is to be *the* system and not being only a part of it.

If we want to incorporate COTS software into a system, we have to ensure that the overall system state (data, control) is maintained consistently with that of the COTS software. Therefore, we have to "observe" the COTS software and then forward all events to the system. The system decides if the changes are relevant and updates itself if necessary.

There are two ways in doing that: 1) batch notification and 2) incremental notification.

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 COTS software may pose major synchronization problems (e.g., complete re-export of all design data from Rational Rose).



Batch notification is computationally expensive and thus only then feasible if the integration between the COTS software and the rest of the system is loose. 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
- Detecting changes and forwarding them does not require manual overhead

Scoping Change Detection

There is a trivial albeit computationally infeasible approach to identifying changes in COTS software by caching its state and continuously comparing its current state with the cached state. This approach obviously does not work 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 identify changes in COTS software smarter it has to be understood how and when changes happen.

Take, for example, IBM Rational Rose. In Rose a user may create a new class 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 can be changed from its default and class features such as methods and attributes may be added. These changes may 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.

Change Detection in Time and Space

Changes are detected by comparing a previous state of a system 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. Our approach therefore uses the programmatic interface of the COTS software to elicit and cache state information. For example, in IBM Rational Rose this may include all its 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 only change information for class diagrams are needed. Thus, it is not necessary to cache and compare other diagrams such as sequence or use-case diagrams.

Basic Change Detection

After every mouse/keyboard event, we ask Rose via its programmatic interface what elements have been selected 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., inhouse developed components) about this difference. Thus, our approach notifies other components on the behalf of the COTS software. If we find a difference, we also update the cache because we want to find and report a difference 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.

This approach detects changes between the cached and current state. But 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 we would have cached it earlier). Thus, we notify other components of the newly created element and create a cached element for future comparisons. 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 can only be detected after de-selection (i.e., a deleted element is a previously selected element that was deleted) and 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 correct always. Certain changes to selected elements may trigger changes to "adjacent", semantically-related 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 and it also causes a change to class Y (i.e., it now does not have a relationship to X any more) although the latter are never selected.

There are two ways to handle the ripple effect. The easiest way is to redefine selection to include all elements that might be affected by a change. For example, if a class is selected we could define that also all its relationships are selected. Then change detection will compare 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 be computational expensive.

The 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 can implement 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 the ripple effect cover most change detection scenarios. However, there may be exceptions that cannot be handled in a disciplined manner. We found only few scenarios in Rose that had to be handled differently.

For example, 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 can. 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.

Consistency between Different Domains

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 instance, the above example integrated Rational Rose design information with UML-compatible design information and both are conceptually similar. 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 is motivated by our need to have a domain-specific component model, called the ESCM (Embedded Systems Component Model) (Schulte 2002), 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 more 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. Batch transformation was used to create a, initial, cached copy of the Rose data (transformation also re-interpreted the Rose data into UML 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. A simplification is to implement change detecting with the possibility of reporting false positives (Rose change does not affect the ESCM) but the guarantee of not omitting true positives (Rose change affects 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 it only lead to some unnecessary but harmless synchronization tasks. The ability to relax the quality of change detection to also allow false positives strongly improved computational complexity.

There are two simple strategies in doing change detection with false positives: 1) we delete all ESCM elements which could be affected through a change in the COTS tool and simply re-transform all deleted elements or 2) we compare all possibly affected elements with the cached data and re-transform only the changed ones. The first strategy is the cheapest but produces more false positives than the second strategy. The more detailed discussion of these strategies is out of the scope here.

Conclusion

Consistency between commercial-off-the-shelf software (COTS), their wrappers, and other components is a pre-condition for a working COTS-based system. Our experience is that it is possible to get notification messages about changes from GUI-driven COTS software even if the COTS software vendor did not provide a (complete) programmatic interface for doing so. This paper briefly discussed several strategies for adding change detection mechanisms to COTS software.

1. Egyed, Alexander and Balzer, Robert. Unfriendly COTS Integration - Instrumentation and Interfaces for Improved Plugability. Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)San Diego, USA; 2001 Nov.

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