Adapting COTS Products

The Fine Line between Development and Maintenance

David Wile, Robert Balzer, Neil Goldman
Teknowledge Corporation, USA
{bbalzer, ngoldman, dwile}@teknowledge.com

Alexander Egyed
Johannes Kepler University, Austria
alexander.egyed@jku.at

Marcelo Tallis
University of Southern California, USA
marcelotallis@gmail.com

Tim Hollebeek
BitArmor Systems, Inc., USA
timhollebeek@gmail.com

Abstract—**COTS products can play various architectural roles in software systems: as interfaces to problem-specific functionality, as components that provide such functionality itself, and as intermediary connectors and components in more complex systems. In doing so, COTS products impose their own, unique constraints on organization and functionality. Over the last ten years, we have gained considerable experience with adopting, adapting, and living with the limitations of COTS products. Our goal was to adapt the COTS product to make it fit the application rather than adapting the application needs to make them fit the COTS product – thus, in essence, adapting the COTS product without access to its source code or documentation (a unique form of maintenance). We report on a large set of experiences involving eight COTS products and a wide range of COTS-Based Software Systems – most of which were done with and for industrial partners or government agencies. This experience report attempts to both give a feeling for how applications can be augmented with such COTS interfaces and also tries to tease out the specific architectural issues that anyone adapting COTS products is certain to face.**

I. INTRODUCTION

Today, when beginning to design a new product, one is faced with a rich set of choices for how to implement the desired functionality. Often, choices are predetermined by organizational, social, and educational preferences or requirements – such as the programming language to use, the development methodology, the platform the application needs to run on, the amount of effort to spend on certain aspects of the project, or some of the interfaces with existing functionality, to name a few. And often, the desired functionality may be constrained by such choices as well. But even once a language has been determined and a platform chosen and once appropriate functionality has been specified, considerable flexibility remains.

Today, the adoption of Commercial of the Shelf Products (COTS components) is a reasonable implementation option. The two primary benefits are first, development of application support fabric is best left to experts at developing support fabric, not experts in the application. Second, carry over from other usage of the same COTS tool in different contexts shortens the learning curve allowing new users to concentrate on new functionality.[1] However, adopting a COTS component resembles maintenance in many ways. One often needs to understand a COTS component's internal states and data without access to source code or documentation. And one often needs to manipulate the COTS component to make it fit the application (rather than making the application fit the COTS product).

Ironically, since adopting COTS products resembles software maintenance, maintenance-related activities apply even to the development of new software systems. This paper discusses the unique constraints that COTS product integration imposes on software system development and subsequent maintenance (the more obvious maintenance problem). The needs of COTS components and how they affect systems and their development/maintenance (platform, methodology, process, functionality) are not always reconcilable with organizational and functional needs. Moreover, the capabilities of a COTS component may outright contradict the needs of the architecture (form, structure, interaction, extent and degree of data persistence, and data and GUI synchronicity). This paper reports on 10 years of experience we gained in adopting, adapting, and simply living with the limitations of COTS components. We will demonstrate that COTS components should not be perceived as unchangeable entities, but can be adapted without access to source code; their GUIs, programmatic APIs, and even behavior can be made to better fit the organizational, functional, and architectural requirements of a software system. It is important to emphasize the extent to which COTS products can be made to fit the application rather than compromising the application to fit COTS products.

II. CHALLENGES OF COTS INTEGRATION

While adapting these components we did not care to identify whether the causes of our problems were with the functionality of the COTS products, their architecture, or in

[1] Of course, there are more "human-centered" issues that may easily dominate the decision to adapt or not but ignored here.

26th IEEE International Conference on Software Maintenance in Timişoara, Romania

978-1-4244-8628-1/10/$26.00 ©2010 IEEE

fact the functionality or architecture we desired, so it is somewhat difficult to button-hole the problems easily. Before attempting to draw specific conclusions about architectural mismatch issues we deal with overarching issues, such as how to interact with users, how to maintain necessary information about the application's state, how to report erroneous or anomalous situations, as well as how to provide an intuitive model of how the product behaves. By doing so, however, we hope to help future COTS adaptation efforts of ourselves and others to go somewhat more smoothly, since identifying these mismatches and understanding the various solutions we attempted may simplify them.

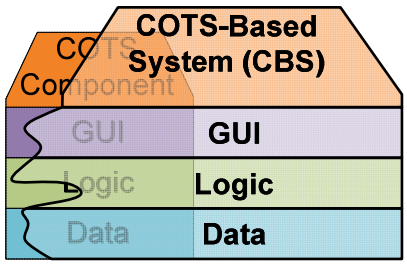


Figure 1. Integrating a COTS component into a CBS requires GUI, Logic, and Data Integration

Although others have identified architectural mismatches as serious impediments to adaptation [1][2] this paper provides a different perspective from the outset by focusing on the maintenance of consistency of several models: the data model of the COTS product itself, the application data model, and the persistent model or models of each. In addition with COTS components that have a GUI, there are issues concerning the information displayed by the GUI versus information available from the COTS API (its "internal" model). And there is a distinction between the services (logic) a COTS component provides versus the ones the CBS needs. Two kinds of data are maintained in the application's data model: information needed to augment the GUI and problem domain-specific data. Finally, there will usually need to be persistent models of all four of these data sources.

III. Case Studies Motivate COTS Challenges

To get a sense of the breadth of possible uses of COTS products for infrastructure support, as well as the variety of architectural roles they can play, aspects of several of the tools we designed over the last ten years will be introduced in the discussion below. Many of our tools have relied on an adaptation of Microsoft PowerPoint for composing briefings and presenting them, called the Briefing Associate [3]. We have also made strong use of IBM Rational Rose to support software modeling – most notably to support the modeling of product line architectures. Furthermore, we have incorporated MS Word, MS Access, and MS Excel in various applications from information assurance to deductive reasoning to census form design [4]. And, we have integrated with a range of other applications, such as Mathwork's Matlab/Stateflow, IBM Rational Software Modeler, MS Outlook and MS Internet Exporer.

Our experiences with integrating these many COTS products were positive. In all cases, the COTS products were chosen based on their wide availability and the users' familiarity. Often, the intent was to leverage from the existing COTS product and extend its capabilities to do more – in essence, to use the COTS product as a component in a COTS-based (software) system (CBS) [5]. In all cases, our integration with these COTS components was so tight that the user of the COTS component was not easily able to distinguish our product extensions from the natural look-and-feel of the COTS component itself. However, this tight integration also caused us the most problems. Our extensions to the COTS component often needed to interact seamlessly with all existing system features in real-time. Unfortunately, none of the COTS components we integrated with provided an API for tracking user changes. So, while the use of a COTS component in a COTS-based software system can significantly accelerate development and provide the user with a familiar interface, it also introduces unique challenges. In this paper, we discuss the issues most likely to be encountered in future efforts to build COTS-based software systems, and our solutions. In fact, many of these will need to be faced during the integration of COTS components into software systems.

IV. 4. Problems and Concerns

Most everyone is aware of program library support for the different look-and-feel paradigms supported by specific platforms. And there is good reason not to stray too far from the familiar paradigms for such activities: each new product has a learning curve associated with it, and the more diversions from conventions understood by clients, the more noise to divert attention from the primary purpose of the product.

A. Active versus Passive COTS Components

However, the look-and-feel provided by the OS is only a small subset of the user's background knowledge. Today, one can leverage the user's familiarity with a wide variety of tools used by all computer users in their everyday activities, like the modern COTS products used for word processing, document preparation, briefing creation and presentation, graphical editing, photographic editing, web form design, database design and storage, and spreadsheet construction. Even if one recognizes the similarity between, for example, composing email and word processing, few designers are aware of the extent of the openness of COTS products and the ability, for example, to embed a word processor from within a proprietary product. Many of these COTS products have accessible, documented (more-or-less) application program interfaces (APIs) that allow them to be used by other programs, not just through the GUIs provided for the tools. Moreover, these tools can often be used to provide a considerable portion of the support infrastructure needed for idiosyncratic product development. By relying on the design of this support infrastructure by experts in GUIs, databases, persistence, graphics, etc., developers can focus on the idiosyncratic part of product design – the novelty of the product itself.

For example, assume one is designing a tool for energy consumption analysis in a factory. Drawing the heat flow diagram for the factory does not just need to "feel something

like" drawing a graph in PowerPoint, it can "be" drawing a graph in PowerPoint. There are a number of ways one can adapt a COTS product such as PowerPoint. The most basic can be described as "passive," where our energy consumption analysis tool is simply given a PowerPoint presentation as its input and the tool uses the PowerPoint API to query the state of the design – interpreting specific shapes and connectors in problem design terms, runs its analysis, and perhaps reports it in a summary window unrelated to PowerPoint. However, there are more "active" ways of using the tool. First, one could design special toolbars for use by the user in PowerPoint to facilitate the drawing activity. One might even write an analyzer that warns the user in real-time that the drawing is ill-formed when the user connects a heater intake to an air conditioning vent, for example. The flow analysis could be reflected onto the diagram, perhaps using colors or line thicknesses to illustrate heat or cold concentrations. One might even animate the diagram with arrows of different thicknesses representing intensities, appearing to flow across the diagram.

Each of these example usages illustrates the power of the idea of using COTS components: in not needing to deal with the myriad details involved in drawing graphs and flow diagrams; in writing connectivity analysis in problem domain terms rather than graphical terms; and in not needing to design graphical or animation output primitives. However, COTS products are typically not designed or built with the thought of using them in COTS-Based Systems. It takes more than a simple API for accessing a COTS' data store to enable the kinds of integrations outlined above. In the following, we summarize the consistency issues of data integration and the axes of concerns when integrating COTS products as components into a CBS.

B. Data Models

One of the most significant issues in dealing with COTS components in a CBS is the issue of data integration. In the CBS, there are two kinds of data models: that of the COTS product and that of the desired application as reflected in the overall CBS. The COTS product is never aware of the overall system state (it was built without knowledge of its use):

- COTS data model
  - Information displayed by the COTS GUI
  - Information available from the COTS API
- Extended data model of the CBS
  - Information augmenting the GUI
  - Problem, domain-specific data
- Data Persistency (COTS data is stored separately from CBS data)

Overall system state often requires that the COTS data adhere to certain constraints. It is important to distinguish between the COTS GUI data (i.e., the data that is visible via the COTS GUI), and the COTS API data (i.e., the data that is accessible via the COTS programmatic interface). Interestingly, the two are not always the same. Moreover, updates to the one may not always (or instantly) update the other (i.e., a user change in the GUI may not immediately be obvious through the API or a data change made through the API may not be visualized in the GUI). There is also an issue with regard to data persistency. COTS products typically have their own means of storing data. In the likely case where the CBS extends the COTS data model, data persistency requires synchronizing multiple data files with concomitant synchronicity issues.

C. Axes of Concerns

It is easy to become overwhelmed when thinking of adapting COTS products for use in an application CBS. We have run into problems in all of the following areas:

- Adaptability of the COTS product: It is rather difficult to adapt unless the API to the COTS product is exposed. Although we have used "wrapping technology" to get around this, it tends to be fragile and dependent on component versions.
- Naturalness of transformation between models: Hopefully the mappings will be largely straight-forward, otherwise the product being adapted may be inappropriate. However one is certain to run into artificialities and inconsistencies, requiring complex transformations in some cases. One must pay particular attention to how easily the application's and the chosen COTS product's consistency requirements are maintainable.
- Naturalness of transformation of operations between models: All of the above concerns apply to the mappings between actions in the models as well – as in reflecting modifications from one model through to others (i.e., the ability to define operations that seamlessly flow across COTS/CBS boundaries).
- Locus of control – API, GUI, Application: Although others have emphasized this aspect as a prime area for potential architectural mismatch, there are some idioms of COTS products that can make the problems a little more approachable. Synchronization and transaction control problems often revolve about the granularity of interception of user activated events. Unfortunately it is often difficult to detect such events even in products whose APIs are accessible!

V. CHALLENGES AND THEIR CHARACTERISTICS

In the following, we discuss the data model issues and axes of concerns on a range of CBS projects we worked on over the past 10 years. It is worthwhile noting that we had to adapt or augment all COTS components we used to enable the kinds of integrations needed. Fortunately, significant COTS product adaptation and augmentation are possible despite lack of access to source code and even documentation.

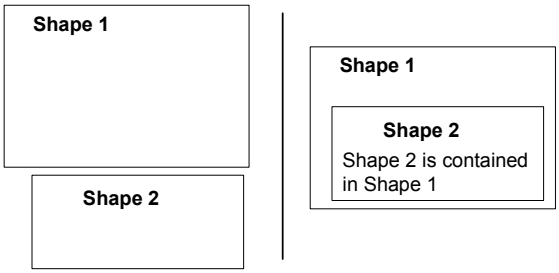
COTS and CBS Models Must Be Similar

Since we used COTS products only to support our special purpose functionality, the biggest challenge we faced was to abstract the internal COTS data model and the transitions between internal application states into a corresponding data model and transitions for the application we were developing. Conversely, transitions between our internal states needed to trigger updates to the corresponding elements of the COTS data model.

We found that tight integration between our extensions and the COTS products was generally only effective if the mappings between the COTS internal model and the application model were simple and natural. We generally required both the ability to read data from the COTS product and the ability to modify this data. Fortunately, most of the COTS products we worked with had very rich APIs; the worst were still accessible through XML or HTML.

Sometimes the translation between the data models of the COTS product and the extended application was straightforward, though there were exceptions. For example, we used IBM Rational Rose to implement a UML 1.3 compliant modeling tool. While Rose was the premiere UML modeling tool at that time, its API reflected earlier design notations because of backward compatibility. The data model of our new UML-compliant tool mapped reasonably closely to the original Rose data model, but translations were required. This was also the case where we used PowerPoint's connectors and shapes to model the relationships between ontologically characterized data in the Briefing Associate.

Unfortunately, not all translations were easy. For example, on one occasion in PowerPoint we wanted the ontology-based data structure relationships to derive from the spatial embedding of one shape in another, specifically to represent a containment hierarchy (see figure below). Unlike with shapes connected via connectors (arrows) unfortunately, PowerPoint had no concept of shape embedding in its API. Thus, the translation between the data structures of the original and the extended PowerPoint needed to recognize such containment based on geometrical analysis.



Although such recognition is not difficult to program, the tight integration between COTS products and our code often required instant awareness of changes. As such, it was necessary to re-evaluate such semantic relationships in real-time while the user was making changes through the GUI of the COTS product (i.e., moving a shape in and out of the other shape). This need for instant user tracking and re-evaluation required very detailed knowledge on when and how user actions caused changes in the data model of COTS products, which turned out to be the subject of our next major challenge.

Implication: It is interesting to note that what may appear to be a simple model mismatch between the application and the COTS system can become an architectural concern because of the time needed to effect the model transformation. This was a somewhat surprising complication to us.

Detecting Changes through "Selection" Is Tricky

To respond instantaneously to changes to the COTS data model, we were forced to deal with extended data model updates incrementally. The standard practice to accomplish this was to maintain a parallel representation of the extended data model and to map changes to the COTS data model into changes to the extended data model. In order to maintain the consistency between reflected external data and COTS-internal data, we found it necessary to closely observe, even track, the user interaction with the COTS product.

Unfortunately, in this respect we found all COTS products to be rather limited. While all COTS products provided some observable events of user changes, these events did not nearly cover our needs. For example, Rose was willing to notify us of events such as the loading of a model; however it provided almost no notification on changes to the data contained in these models. We thus implemented a mechanism for inferring user changes based on user selection, caching and comparison [6].

In most modern, GUI-based software systems, changes to the COTS internal model happen through selection. For example, in PowerPoint a shape must be selected by a user before its name can be changed or it can be deleted. Fortunately, we found that the API of COTS products did allow us to query for selection. We thus cached properties of COTS internal model elements when the user selected them through the COTS GUI; and then compared the cached properties with the properties of those elements when they were deselected. Changes to these properties implied internal model changes. Indeed, this mechanism accounted for our ability to monitor most GUI-instigated changes. However, there were further issues:

- Selection revealed what GUI data could change but it did not reveal when it might change – we occasionally found it necessary to use low-level, operating system monitoring to trap mouse and keyboard events to infer this timing [7][8]..
- A race condition could occur in situations where the initial caching of selected GUI elements was not quick enough and the user changed some or all of these elements before they were fully cached.

In some cases, the caching and periodic comparison was not feasible due to the large amount of data involved. For example, we found the simple caching and comparison technique to be insufficient to track user changes in MS Word. Rather, a combination of clues was used, gathered from Word's undo stack, cursor locations, specific mouse events, and other sources, in order to narrow down the list of possible changes.

Unfortunately, one could never be sure whether all GUI changes were tracked and identified correctly as there very many ways of changing information through the GUI of a typical software product; sometimes we had to resort to disabling certain GUI actions. Also, while this tracking mechanism sounds simple in principle, it was rather elaborate to implement and to debug.

Implication: The issue here is clear: to use an autonomous component as though it is a subcomponent can require considerable mechanism and may even need to rely on

adapting the user's behavior to, for example, delay further GUI actions until processing is completed. This is not necessarily a deal breaker: users are quite adaptable if the benefits appear to be worthwhile to them.

GUI Customization Is Usually Straightforward

We often found it necessary to augment the GUI of COTS products, for example, by changing menus or the way applications displayed data (i.e., shapes, symbols, sub-windows). As was mentioned above, we often used PowerPoint to provide a graphical interface for problem domains characterized by formal ontologies. In these domains, we used PowerPoint shapes, icons, and relationships among them to emphasize domain-specific concepts. We also used or augmented the PowerPoint menus, popup menus, dialog boxes, and shape attributes to support the description of these domain-specific concepts and to add information to the application model not represented in the COTS internal model.

We found that all COTS products we used lent themselves easily to these kinds of extensions. However, there were limitations, especially when encountering a consistency rule maintained by the COTS component that is not appropriate for the application. For example, we found that Rose prevented the user from making arbitrary connections between shapes. For example, Rose prevented circular inheritance in its class diagrams. This restriction, while perfectly valid for UML models, was invalid for the modeling of product line architectures, our application. Consequently, we were forced to model inheritance differently and less elegantly than a user of Rose might have expected. Likewise, we found that Rose did not enforce certain drawing constraints that we would have liked to have – thus allowing the user to create structures in Rose that were incorrect from a product line perspective (though correct from the perspective of UML). Fortunately, the latter problem was solvable by tracking user changes. When we observed a change, we could evaluate it in real-time and reason about its legality. We could even modify the user action (e.g., undo it) if necessary; or we could block undesired activity.

Implication: This case is similar to the model mismatch above, in that here a consistency rule mismatch ultimately has architectural implications. As a general rule, we found it much easier to augment the behavior of a COTS product to enforce additional constraints but almost always failed to adapt the COTS product to prevent something from happening. The obvious implication for COTS product selection is thus to err on the side of selecting a COTS product that does too little rather than too much.

GUI and COTS Internal Models Often Inconsistent

One would expect that the information depicted in the GUI of a COTS product would always be consistent with the data accessible through its API. However, we found exceptions. For example, MS Word did not update its accessible data instantly with changes (i.e., as new words were typed). Rather there was a delay, suggesting the existence of another data store within Word that was not accessible via its API. This caused problems in our attempt to maintain consistency.

In PowerPoint, we encountered the opposite problem. There, GUI changes made through the API were immediately updated internally and the GUI consequently triggered screen refreshes. Unfortunately, we often needed to make several changes through the GUI at once, transactions in essence. These sequences of changes thus resulted in the flickering of the screen – one flicker per change – which had a rather irritating effect. In order to make such transactions flicker-free, we resorted to low-level interception and blocking of operating system commands related to the refreshing of windows.

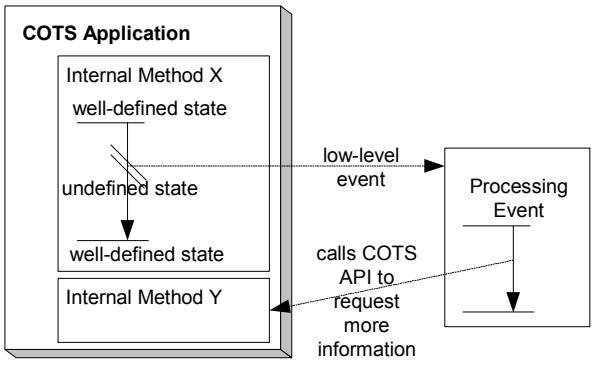
With Rose, we faced another dilemma. There, changes made through the API were at times elaborate and could take seconds to process. Thus, while our code made API calls on Rose, there was nothing to prevent the user from making concurrent modifications through Rose's GUI. For example, it was possible for a user to delete a GUI element that was at the same time updated through the API. This resulted in unpredictable exceptions. To solve this problem, we introduced a locking/unlocking mechanism that blocked user events, such as keyboard and mouse events, while API access was in progress. This blocking was also achieved through low-level tweaking of operating system events.

Implication: The lesson here is that sometimes the internal data/state (architecture) of the COTS product becomes an important issue. The next section discusses an even more insidious manifestation of this issue.

COTS as Black Boxes Confuses Assumptions

Tweaking operating system events or using monitoring tools to track user changes in COTS products had many advantages but it did have one major problem. We relied on low-level monitoring of the COTS product – often via installed wrappers – to look out for pre-defined patterns of activities and raise events once they were encountered. These events typically blocked the execution of the COTS product until they had been processed. Unfortunately, we never knew exactly in what state the COTS product was at the time an event was raised.

We know from traditional software engineering that the state of software systems (not just COTS) is well defined before and after method calls. However, most of the monitoring mechanisms were low-level spying tools that were unaware of method boundaries within the COTS product. Thus, at the time an event was raised, the COTS product was likely to be processing one of its internal methods and was thus in an undefined state (see figure above). This was not a problem if the processing of the event did not affect the COTS product.



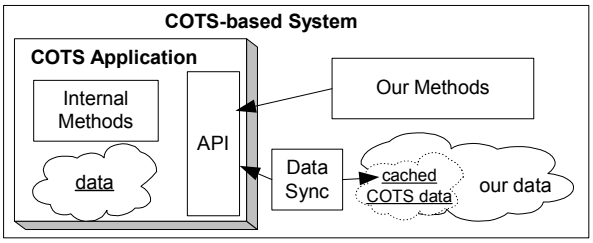
Unfortunately, more often than not the processing of an event required access to the COTS product, to gather more information or to trigger changes. Consequently, methods of the API were invoked concurrently while the COTS product was already in the middle of processing a method and while the COTS product was in some undefined, intermittent state (e.g., perhaps it had changed a pointer, but had not yet changed the back pointer). Situations like these did cause problems and these problems were hard to debug. For example, we had one particularly bad experience with Rose where such tweaking caused the assignment of the same unique id to more than one data element and, consequently, caused abnormal behavior whose effects were only exhibited much later.

This problem not only manifested as inconsistent data state within a COTS product but also affected the control flow, as events that were being processed were interrupted because they made changes to the COTS product that triggered other events. Infinite loops were a not unusual effect.

Implication: In some cases, this problem could be circumvented by not processing the events instantly but rather by queuing them and processing them later. Unfortunately, delaying processing of events made some of our applications vulnerable to race conditions.

Piggybacked or Parallel CBS Persistence

We often dealt with two separate sets of data in our extended COTS products – the data that belonged to the COTS product itself and additional application data needed by our extensions that we built on top of these COTS products. Earlier, we explained that we used caching extensively to mirror the COTS data. This was done to make the data easier to access but also to better integrate this data with our data. Above, we explained some of the challenges we faced in ensuring the consistency between the cached COTS data and the actual COTS data. Here, we talk about how we had to deal with the persistent storage of our data along with the COTS data.



With Rose and PowerPoint we were fortunate that the COTS products supported the use of tagged string attributes (name – value pairs) to enhance application modeling. Both COTS products stored the tagged values together with their own data. Persistent storage was thus handled by writing tagged values to the COTS product and using its internal methods for persistently writing files.

Unfortunately, not all COTS products were this supportive. In case of Word, we found no adequate way of having it write our own data. We were thus forced to maintain distinct data files – one maintained by the COTS product and the other maintained by us. This was problematic in two ways:

- a user could move one file but not the other to a new location, thus breaking the tie between them
- a user could modify one of the files by starting the COTS product as a standalone application (without our code enabled)

Working with persistent storage thus required some discipline on the part of the user not to do either of the above. Windows Alternative Data Streams provide a clean way of solving the first problem, as they allow additional streams of bytes to be associated with files in a way that is invisible to applications that just use the main stream. However, they still will not be recognized, updated, or copied by non-instrumented versions of the application. A possible solution is to instrument the application in such a way that it is always instrumented, but the extra functionality is only enabled when appropriate.

Implication: What may appear to be a single entity, may be split into multiple entities during COTS integration (i.e., instead of one data store, we now have two). Since the COTS product affects architectural decisions, such interplays should be elevated to architectural concerns and modeled as such (e.g., much like a distributed data store)

Insulate CBS from COTS Version Changes

We worked through many versions of COTS products over the course of 10 years. A lesson we ultimately learned was that, since we had no control over the evolution of the COTS products, a seemingly redundant layer of code that characterizes our interface to the COTS API should be used from the beginning. Changes introduced in subsequent versions may then be worked around in this veneer.

First and foremost, which version of COTS product to use was typically decided by our customers. It was also our customers' decision when to upgrade from one version to another. We were fortunate in that we never had to deal with more than one customer at a time and as such we never had to deal with the problem of having customers upgrading (or not upgrading) their software at different times.

All COTS products we worked with tried to maintain upward compatibility. Unfortunately, this did not mean that our applications continued to function correctly after upgrades. In some cases, bugs in the COTS product caused problems, but the addition of new features and fixing of old bugs could also be problematic. For example, we frequently encountered situations where new options were added. Thus it was important that our implementations were very careful about any assumptions they made about the range of possible choices and values.

However, a more serious problem was semantic changes to the API without changing the syntax. For example, we found that at one point a new version of Rose reversed the meaning of a connector used in the GUI. Instead of A calling B the reversal implied B calling A. In PowerPoint, we found that a newer version changed the number ordering on how connectors were attached to shapes.

The most serious issue, however, is when a COTS product is evolved in a way that it becomes incompatible with our

needs. The COTS product may even be discontinued (e.g., Rose). Although it was the premier UML modeling tool, IBM seems to have decided not to support its evolution. The replacement tool they provide offers no backward compatibility.

On the whole, though, the COTS APIs are quite stable, and the amount of effort needed to support new versions is substantially less than what might naively be expected. For example, one of our security products makes extensive, low-level changes to application and operating system behavior, but still only required one or two man-months to be ported from Windows 2000/Office 2000 to Windows XP/Office 2003.

Implications: Shielding an system from version changes is desirable but impossible to ensure. If there is a need to maintain multiple COTS product versions for a CBS (i.e., for different customers) then their architectural variability should be modeled explicitly.

Application and GUI Operations Differ

We previously discussed the need to monitor user changes to the GUI, in particular, to ensure data consistency between the COTS-internal application data and the cached data. However, other forms of user tracking and monitoring became necessary to handle undo, saving, or exiting.

Many COTS products allow the user to undo changes. This ability caused problems. For example, in Excel, the undo stack was emptied after an API call. This feature was annoying to the user but actually an advantage compared to the problems we faced with Rose and Word. In Word, API changes and user changes became part of the undo stack. The user could thus undo changes our code made to Word. Many of our changes came in sequences of steps that were related and should not be undone partially. Unfortunately, Word allowed the user to undo these steps individually. We were able to solve this problem by inserting begin/end markers in the undo stack before/after making changes through the API. Then, we used low-level monitoring tools to observe when the user invoked undo in Word (either through the menu or the keyboard). The event then intervened in the Word undo process to sequentially undo related changes.

With respect to low level monitoring, it was often necessary to recognize certain menu options, GUI elements, and so on. This can be tricky, as internal application IDs are not necessarily stable between versions, and externally visible text is not stable in the presence of internationalization. At this point, we have generally dealt with this issue via ad hoc heuristics, depending on the situation.

In addition to monitoring undo, we often also had to pay special attention to the user quitting the COTS product (i.e., requiring cleanup), the user invoking Save As (i.e., requiring explicit persistent storage), or the user changing a model/document/workbook (i.e., changing the data model). Monitoring these user actions was usually straightforward, but quite tricky at times.

Implication: Once again the concern seems to be not in componentry or connectors, but rather in the definition of events; there is a mismatch between the events supported by the COTS (or even one of its subcomponents) and the apparent events of the application CBS.

Adapting/Augmenting Behavior

It is generally much harder to adapt the behavior of a COTS product instead of augmenting it, because, it requires the COTS product to stop doing something that it is pre-defined to do. In the few cases where adaption was necessary, we resorted to the manipulation of low-level events (e.g., to prevent the GUI flickering with multiple PowerPoint updates as was discussed above). However, we also resorted to more radical measures, for example, by re-developing some of the COTS products' dependent sub-components. Complex software systems typically comprise subcomponents and it is possible to exchange these subcomponents as long as the interfaces remain compatible. For example, in the case of the modeling tool IBM Rational Software Modeler, we generated a new version of the UML data model subcomponent and replaced the existing one with it. This was done because of our need to fully integrate the COTS product's data model with our CBS model to avoid synchronicity issues. The increasing reliance of software systems on sub components (e.g., plug ins, add-ons, DLLs, services) makes this replacement strategy a powerful option. It is however only possible if the development of the replaced sub component is both technically feasible and cost effective.

Testing the Application and COTS Products

While the (re)use of COTS products within the CBS saves a significant amount of development time, it does not necessarily save as much testing time. An adapted or augmented COTS product must be tested extensively; so must all interactions between the COTS product and the remaining applications of the CBS.

Related to testing is the issue of error handling, such as exception handling. Much as a COTS product may not provide all services or data needed for the CBS, it may also lack important kinds of exceptions. Moreover, any recovery that happens as a result of erroneous behavior must be synchronized such that the COTS product and the remaining parts of the CBS remain consistent. We often resorted to wrapping COTS products and these wrappers also explicitly mitigated error handling (i.e., triggering exceptions on the behalf of the COTS product if needed, transforming COTS product exceptions into CBS exceptions, and otherwise synchronizing error handling).

VI. Related Work

As early adopters and adapters of COTS products as serious alternatives to component based design or even less disciplined programming techniques, we were not really influenced in the adaptation techniques by other researchers.[2] However, over the last several years considerable attention has been paid to COTS incorporation [9] and it is worth mentioning how we feel this report distinguishes itself from that body of work. In the

[2] Of course, we were influenced by research in the applications for which the CBSs were designed, but their discussion is out of scope .

COTS area considerable effort has gone into trying to specify functionality of the COTS products required by the embedding CBS [10][11], in capturing the requirements involving COTS products [12], in testing that the COTS products do or continue to do what is desired [13][14][15], and in generating systems that select COTS having appropriate properties [16] [17] [18].

More appropriate to the concerns here are papers detailing adaptation tricks, techniques, and workarounds [19][8] [7]. But especially relevant are those on architectural mismatches, mentioned previously. The only work concerned with mismatches in COTS explicitly is [20] following on the work of [21] [2]. This feature-based classification helps people to choose COTS products that are compatible with architectural assumptions of the embedding CBS. These are grouped into three categories: packaging mismatches characterized by incompatible communication paths, component dependency assumptions, and internal assumption mismatches [2].

It is fair to say that had we had their analysis techniques and found that, for example, we had severe communication mismatches in components we might have been discouraged from using those COTS components. Here we have illustrated how one can overcome mismatches, sometimes quite serious ones, often through use of different architectural idioms and wrappers which in effect map the mismatches away.

VII. Conclusions and Advice

The paper highlighted many issues that surround the adoption of COTS components into software systems. However, as was seen, the adoption of COTS components is often more reminiscent of adapting/evolving the COTS component – an activity that is similar to software maintenance. Furthermore, the adoption of COTS components also imposes unique maintenance issues on the software system thereafter – having to maintain the adaptations of the COTS component and enduring vendor-imposed evolutions of the COTS component. Almost all of the solutions mentioned can be of general utility in software engineering efforts. For example, the need for instant user tracking and re-evaluation; to use low-level, operating system monitoring to trap mouse and keyboard events to infer timing; to use combinations of clues to track users' activities, low-level interception and blocking of operating system commands related to the refreshing of windows; the introduction of a locking/unlocking mechanism that blocked user events; queuing events and processing them later; maintaining distinct data files for persistence; and finally, instrumenting the application in such a way that it is always instrumented, but the extra functionality is only enabled when appropriate.

Hopefully the insights made in this paper help lead to more principled COTS designs in the future –both for development and maintenance.

VIII. References

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

[2] C. Gacek, "Detecting Architectural Mismatches During System Composition," in *PhD Dissertation, University of Southern California, Los Angeles, USA,* 1998.

[3] M. Tallis, N. Goldman, and R. Balzer, "The Briefing Associate: a role for COTS applications in the Semantic Web," International Semantic Web Working Symposium, Stanford, California, 2001.

[4] D. S. Wile, "Supporting the DSL Spectrum. Journal of Computing and Information Technology," *Journal on Computing and Information Technology* vol. 9, pp. 263-287, 2001.

[5] ICCBSS, "International Conference on COTS-Based Software Systems."

[6] A. Egyed and B. Balzer, "Integrating COTS Software into Systems through Instrumentation and Reasoning," *International Journal of Automated Software Engineering,* vol. 13, pp. 41-64, 2006.

[7] R. Balzer and N. Goldman, "Mediating Connectors: A Non-ByPassable Process Wrapping Technology," DARPA DISCEX Conf., Hilton Head, USA, 2000.

[8] N. Goldman, "Smiley-An Interactive Tool for Monitoring Inter-Module Function Calls," 8th International Workshop on Program Comprehension, Limerick, Ireland, 2000.

[9] A. Egyed, H. A. Müller, D. E. Perry, D. B. Smith, and S. R. Tilley, "Summery of the 2nd International Workshop on Incorporating COTS Software into Software Systems (IWICSS): Tools and Techniques," in *29th International Conference on Software Engineering*. Minneapolis, USA, 2007, pp. 142-143.

[10] E. Kesseler, "Assessing COTS software in a certifiable safety-critical domain," *Information Systems Journal*, vol. 18, pp. 299-324 2008.

[11] M. A. Copenhafer and K. J. Sullivan, "Exploration Harnesses: Tool-Supported Interactive Discovery of Commercial Component Properties," 14th International Conference on Automated Software Engineering, Cocoa Beach, USA, 1999.

[12] J. P. Carvallo, X. Franch, and C. Quer, "Requirements engineering for COTS-based software systems," 2008 Symposium on Applied Computing, Fortaleza, Brazil, 2008.

[13] L. Mariani, S. Papagiannakis, and M. Pezzè, "Compatibility and Regression Testing of COTS-Component-Based Software," 29th International Conference on Software Engineering, Minneapolis, USA, 2007.

[14] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "A Lightweight Process for Change Identification and Regression Test Selection in Using COTS Components," 5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems, Orlando, USA, 2006.

[15] H. Reza, S. Buettner, and V. Krishna, "A Method to Test Component Off-the-Shelf (COTS) Used in Safety Critical Systems," 5th International Conference on Information Technology: New Generations, Las Vegas, USA, 2008.

[16] M. Grechanik, "Finding errors in components that exchange xml data," 22nd Intern. Conference on Automated Software Eng., Atlanta, USA, 2007.

[17] V. Cortellessa, I. Crnkovic, F. Marinell, and P. Potena, "Experimenting the Automated Selection of COTS Components Based on Cost and System Requirements," *The Journal of Universal Computer Science*, vol. 14, pp. 1228-1255, 2008.

[18] R. Land, L. Blankers, M. R. V. Chaudron, and I. Crnkovic, "COTS Selection Best Practices in Literature and in Industry," 10th Intern. Conference on Software Reuse, Beijing, China, 2008.

[19] M. E. Shin and F. Paniagua, "Design of Wrapper for Self-Management of COTS Components," 19th International Conference on Software Engineering & Knowledge Engineering Boston, USA, 2007.

[20] J. Bhuta, C. Mattmann, N. Medvidovic, and B. W. Boehm, "A Framework for the Assessment and Selection of Software Components and Connectors in COTS-Based Architectures," 6th Working Conference on Software Architecture, Mumbai, India, 2007.

[21] A. Abd-Allah, "Composing Heterogeneous Software Architectures," in *PhD Dissertation, University of Southern California, Los Angeles, USA,* 1996.