Automated Software Engineering, 13, 41–64, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Integrating COTS Software into Systems through Instrumentation and Reasoning

ALEXANDER EGYED aegyed@teknowledge.com
ROBERT BALZER balzer@teknowledge.com
Teknowledge Corporation, 4640 Admiralty Way, Suite 1010, Marina Del Rey, CA 90292, USA

Abstract. Commercial-off-the-shelf (COTS) software tends to be cheap, reliable, and functionally powerful due to its large user base. It has thus become highly desirable to incorporate COTS software into software products (systems) as it can significantly reduce development cost and effort, while maintaining overall software product quality and increasing product acceptance. However, incorporating COTS software into software products introduces new complexities that developers are currently ill equipped to handle. Most significantly, while COTS software frequently contains programmatic interfaces that allow other software components to obtain services from them on a direct call basis, they usually lack the ability to initiate interactions with other components. This often leads to problems of state and/or data inconsistency. This paper presents a framework for integrating COTS software as proactive components within a software system that maintain the consistency of the state and data they share with other components. The framework utilizes a combination of low-level instrumentation and high-level reasoning to expose the relevant internal activities within a COTS component required to initiate the communication needed to maintain consistency with the other components with which it shares state and data. We will illustrate these capabilities through the integration of IBM's Rational Rose into a design suite and demonstrate how our framework solves the complex data synchronization problems that arise from this integration.

Keywords: COTS-based software system, COTS integration, change notification

1. Introduction

As a result of their large user base, Commercial-off-the-Shelf (COTS) software systems usually have stable interfaces (APIs) and are reasonably reliable. The need to satisfy a large and diverse user base makes COTS software very generic and functionally powerful with the added advantage that they are widely understood and accepted. Since COTS software also tends to represent large pieces of code, their reuse can significantly reduce development cost and effort (Boehm et al., 2000). These features make COTS software very attractive reuse targets in the wake of exploding software development costs.

We define a COTS-based system to be a software system that includes COTS software (Brownsword et al., 2000). 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 (Boehm et al., 2002; Morisio et al., 2000). There are many advantages in doing so but the lack of source code requires that the reuse of COTS software be treated differently than traditional code reuse (Abts

and Boehm, 1996; Boehm et al., 2000; Morisio et al., 2000). COTS software cannot be tailored from "within" by modifying its source code. Instead, changes must be imposed from the "outside" via wrappers or glue code (Boehm and Abts, 1999; Egyed et al., 2000). Incorporating COTS software into COTS-based systems is risky because the lack of source code makes it very hard to work around deficiencies (Garlan et al., 1995; Morisio et al., 2000). "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." (Boehm et al., 2000)

Abts et al. (2001) identified four primary sources of effort due to COTS-based software development: assessment, tailoring, glue code, and custom application code construction. These sources of effort are also sources of risks during COTS-based software development. Assessment is the evaluation and selection of viable COTS software; tailoring and glue code are the activities associated with integrating COTS software into systems; and custom application code construction is the development of additional, required functionality not covered by the integrated COTS software. Their perspective is supported by data collected from developing several dozens COTS-based systems (Boehm et al., 2002). This data shows that COTS reuse has enjoyed a steady growth in the past years (in some domains) but warns that it does not come without a price. Inappropriate COTS reuse can negate all its benefits and even result in project failure (Garlan et al., 1995; Sedigh-Ali et al., 2001).

The emphasis of this paper is to explore the technical feasibility of integrating COTS software into software systems in cases where the COTS component(s) share state and/or data with other components and the consistency of that state and/or data must be maintained. For example, if a user manipulates the COTS software through its native user interface then these activities typically remain unnoticed by the software system it is integrated with. Consequently, data and state synchronicity problems may arise. The technique presented in this paper is thus useful when the COTS software does not expose changes to the state and data it shares with other components.

This integration problem poses the challenge of how to augment COTS software with notifications of its "internal" changes (data and state) when those changes affect the software system it is integrated with (the COTS-based system). The main difficulty of this integration problem is thus to alter the behavior of the COTS software (e.g., make it communicate information about internal changes) without altering the COTS software itself.

This paper proposes an architectural framework for observing internal activities within COTS software to communicate these activities to other software components proactively. Internal activities can be observed by monitoring how the COTS software is manipulated from the outside (e.g., user interactions) and then selectively query the COTS software about the internal effects of these manipulations. This solution does not require changes to the COTS software itself and, from the perspective of the software system, the COTS software becomes proactive because the framework instigates communication on its behalf to inform other components of the changes made to shared state and/or data.

As an example, consider the integration of a model consistency checking component with a model design component. The stated requirement is that consistency checking should evaluate model design changes only. That is, consistency checking should be smart enough to not re-evaluate portions of the model that have not changed but instead

re-evaluate only the changed portions (i.e., to reduce the computational cost). Further consider that the model design software is the COTS software product IBM Rational Rose (a CASE tool that supports modeling in the Unified Modeling Language (UML) (Booch et al., 1999)) while the consistency checking software is an in-house developed software component. The integrated system should thus function in a way where the model design component will forward changes of its design model to the consistency-checking component to trigger re-evaluations. Unfortunately, IBM Rational Rose has a very limited mechanism to notify other components of changes. It obviously does not understand the needs of our component and it will not notify the consistency-checking component of all required model design changes. Rational Rose thus cannot be used, by itself, to satisfy our system requirement. The framework presented in this paper will passively observe Rational Rose to identify its internal model design changes. The framework, tailored to the needs of our system, will then forward these observed design changes in Rational Rose to the consistency-checking component to trigger its re-evaluations of the shared design model. Rational Rose is neither aware of the framework observing it nor is it aware of the framework instigating communications on its behalf. Rational Rose, together with our framework, now satisfies the system requirement to forward design changes to the consistency-checking component.

We see our framework as augmenting COTS software with added behavior without changing the COTS software from within (i.e., no source code is available). Moreover, the COTS software is augmented to satisfy the requirements of the system it is being integrated with. For example, we will demonstrate how our framework augments several COTS software products (1) to elicit notifications of changes to their data for maintaining data synchronicity between them and other components and (2) to initiate service requests to other components in response to user activities within the COTS software.

We will illustrate the use of this COTS integration framework on Rational Rose and present results of integrating two other, large-scale COTS software products, Matlab/Stateflow and Microsoft PowerPoint. We will demonstrate that COTS software can be augmented to communicate internal changes to other components proactively. We will also describe the technical reasons why these types of COTS integration may fail without our integration framework (e.g., lack of usability, scalability, and reliability). Our framework will work for COTS software integration projects where (1) the data within a COTS component undergoes internal changes in response to external stimuli (e.g., user input) and (2) those changes are restricted to an identifiable subset at the time of the stimuli (e.g., selected data). The former defines when changes happen (e.g., mouse click) and the latter defines where changes happen (e.g., selected element). Many COTS software products satisfy the above conditions. To demonstrate this, we have applied our framework on several major products (e.g. Egyed and Wile (2001) and Tallis and Balzer (2001)) to date.

It must be noted that this paper does not propose a new process for COTS-based software development. Several such processes exist (Boehm et al., 1999; Brownsword et al., 2000; Morisio et al., 2000). Also out of the scope of this paper is COTS assessment which is primarily a risk management activity (Boehm, 1989; Lawlis et al., 2001; Maiden and Ncube, 1998) that precedes software development. Several solutions for COTS assessments are described in Dean and Gravel (2002). This paper solely investigates technical issues for solving the problems outlined above.

Section 2 discusses the pros and cons of integrating reactive and proactive COTS software. Section 3 then introduces our infrastructure for converting reactive COTS software into proactive ones. It defines the roles of mediation, translation, instrumentation, and reasoning to support COTS software integration. Section 4 describes an implementation of our infrastructure for Rational Rose and Section 5 describes three integration scenarios where only one of them (the augmented proactive Rational Rose) satisfies all functional and quality integration goals. Section 6 then discusses other integration scenarios and relevant issues. Section 7 concludes this paper.

2. Reactive and proactive COTS software

COTS products generally assume that they are an independent system rather than a component in some larger system. It is hard to integrate such software because it has no knowledge of the role it is supposed to play in the context of the larger system. Whereas software components (figure 1(a)) are typically capable of both responding to requests (reactive) and initiating requests on their own (proactive), COTS software (figure 1(b)) is generally only capable of responding to requests. This restriction severely limits its reuse.

COTS software typically provides three types of services: (1) logic/functionality, (2) (persistent) data handling, and (3) user interface. If COTS software is reused as part of COTS-based systems then this typically implies a need of managing its services reactively or proactively. This section discusses the pros and cons.

2.1. Reactive COTS software

The most commonly attempted (traditional) integration of COTS software is to have the COTS software accessed by in-house-developed software. In this scenario, COTS

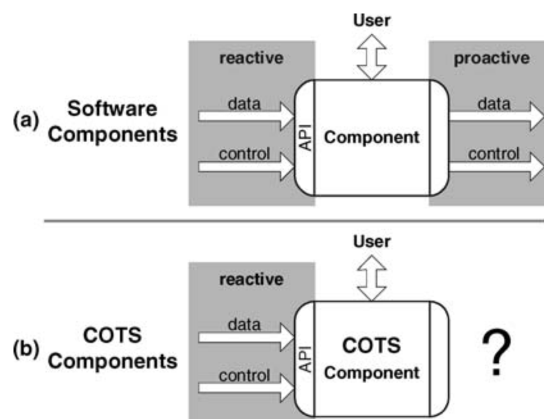

Figure 1. Software components respond to requests of other components but they also initiate requests to others (top); COTS components generally only respond to requests since they have little to no knowledge of their new environment (bottom).

software is used as a reactive, service-providing (COTS) component where "reactive" implies that the COTS component is not required to have any knowledge about the surrounding system that interacts with it (whether they are in-house or other COTS components). It is the nature of reactive components not to initiate interactions with other software components but instead to wait for service requests. From the perspective of the overall COTS-based system architecture, it thus appears as if integrated COTS software is dormant unless it is instructed to do something.

To allow COTS software to be integrated reactively into a system, the COTS software vendor has to provide a programmatic interface (e.g., API-application program interface) that facilitates access to its services. The interface is then used by other software components to interact with it (figure 1(b)). Usually, the interface provides (1) data access for reading and writing the COTS data store and (2) control access for triggering some form of COTS processing. Many COTS vendors supply their COTS software with at least a partially complete interface for data and/or control access. Consequently, COTS software can be integrated as reactive components into COTS-based systems with relative ease. Databases and web servers are typical examples of reactive COTS components.

2.2. *Proactive COTS software*

Most COTS software only initiates 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 (e.g., recall the example in the introduction). 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 of shared state and data from such a COTS component in a system while it is being manipulated by a user. The challenges are:

(1) *Data Inconsistency*: Data captured in COTS software may have to be consumed by other components in a system. If a user manipulates the data within a 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 thus get lost if they are done through the user interface of the COTS software.

In an ideal world, COTS software could be configured to notify other components of relevant internal changes (data and state). In such an ideal world, the COTS software would become 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 section will discuss how this can be accomplished using a combination of instrumentation and reasoning.

3. Augmenting COTS software

This section will show how to augment reactive COTS software into proactive software components with respect to the system they are integrated with. Since it is generally not possible to change COTS software from "within" (no source code is available), we augment its behavior from the "outside." Our approach is most useful in cases where (1) internal activities within COTS software are triggered through outside stimuli (e.g., user input) and (2) the desired proactive behavior of the COTS software is in response to its internal activities (e.g., to notify others of a change).

Figure 2 depicts our infrastructure for augmenting COTS software schematically. The center of the figure holds the actual COTS software. Since no source code is available, it cannot be changed from within. Instrumentation is used to monitor outside stimuli directed towards the COTS software (shaded frame around the COTS software). For example, we use instrumented wrapper technology (Balzer and Goldman, 1999) to observe interactions between a software component and its environment (e.g., user interactions, requests from other components). 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 API) to infer what internal changes this activity caused.

For example, instrumentation may indicate that a delete key was pressed while the COTS API may reveal that a piece of data was selected at that time, but no longer exists. The Reasoning component concludes that the selected element was deleted. The Reasoning component also initiates notifications to other software components of the

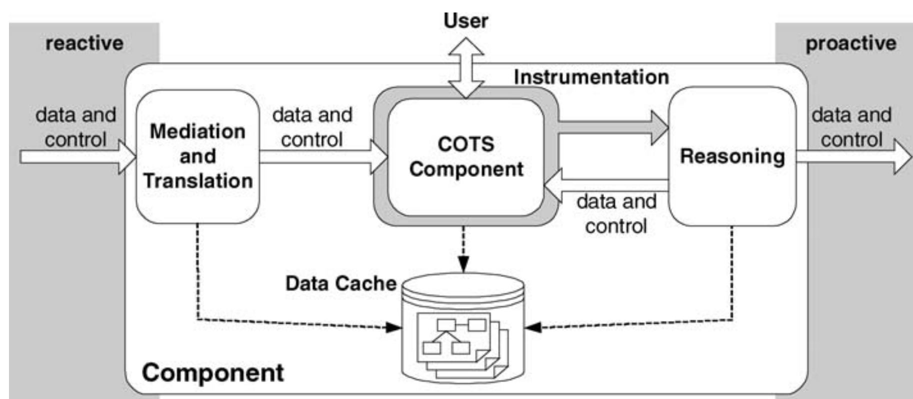

Figure 2. Augmenting a COTS component from the outside through mediation, Translation, Instrumentation, and Reasoning.

system on behalf of the COTS software to inform them of relevant changes. Our framework also provides an optional, alternative interface that may be used to mediate and translate the appearance of the COTS software if required. Other components can then use the alternative interface instead of the COTS native interface. An optional data cache may be required if knowledge of internal activities within a COTS component must remain available for later use. Each of these components are described in the following subsections.

3.1. Instrumentation

User interfaces and programmatic interfaces (API) are typically the only means of accessing and manipulating COTS software. They provide access to COTS data and services in a form that was deemed appropriate by the COTS software designers. Reusing COTS software within software systems requires open, unrestricted access to it. The lack of openness is a severe inhibitor to COTS reuse but can be eased. This section discusses a variety of techniques that have been developed to investigate and observe (COTS) software.

Hooking is a technique for observing and manipulating the interaction between the operating system and COTS software (Balzer and Goldman, 1999). Since the operating system is the core of virtually all communication between the COTS software and the outside world (user), observing and manipulating communications reveals detailed records of COTS behavior. This includes user interface activities, interactions with devices and system libraries. In particular, file access, network communication, user input/output, and interactions between sub-components of software products (including COTS software) are channeled through these hooks. Hooks receive this information and may either passively observe it or actively manipulate it.

Alternatively to hooks, (COTS) software may be changed through its binary representation. Given enough understanding of its binary code, it could be "patched" to replace, delete, or add new functionalities. This form of access, however, requires low-level familiarity with its machine code.

These two instrumentation techniques differ in their invasiveness. While hooks are placed at the interfaces to COTS software and only affect data and control flow to and from them, patches change the actual software. Both techniques have respective benefits and drawbacks and have in common the problem that they may be obsoleted by new releases. There may also be legal implications that have to be considered. Nonetheless, both techniques provide access to otherwise inaccessible data and control information.

Fortunately, instrumentation is only needed to the extent that the required COTS services are inaccessible. We have used hooks to implement all our instrumentation needs (e.g., for Rational Rose, Microsoft PowerPoint, and Mathwork's Matlab) (1) to observe the occurrence and type of user events such as keyboard strokes or mouse clicks, (2) to block user events if desired, and (3) to observe low-level activities such as shutting down the COTS software.

For this instrumentation to be effective, typically, multiple observations have to be combined to infer desired information. Thus, it is possible to derive higher-level, complex, and system-relevant information about COTS software based on simple, low-level

instrumentation. This is the responsibility of the Reasoning component and is described in the next subsection.

3.2. Reasoning

The Reasoning component uses instrumentation and programmatic interfaces to COTS software to infer the internal activities of that component. Reasoning also instigates communication to other components of the COTS-based system on behalf of the COTS software. This is simple when the COTS software externalizes its internal activities through its programmatic interface. The Reasoning component would then periodically request the latest internal activities from the COTS software and initiate change notifications to other components when required. We found that COTS software rarely externalizes all the relevant internal activities required for maintaining the consistency of the state and data being shared with other components. In cases where it does not externalize the relevant internal activities, we apply our instrumentation technologies to infer when these non-externalized internal activities might have occured. Thus, our technique is only applicable to COTS software where these non-externalized internal activities occur in response to some form of external stimuli. The Reasoning component then combines the observed external stimuli, and the observable internal state of the COTS product before and after the external stimuli to infer whether the non-externalized internal activity actually occurred and if so, what changes it caused.

It must be noted that the COTS software typically is not aware that it is being observed and manipulated. Also, other software components typically are not aware that the communication does not originate from the COTS software itself but from the Reasoning component. The Reasoning component thus augments the COTS software with *proactive behavior* that can be tailored towards a particular system. Typically, a particular system does not require access to all internal activities within the COTS software and the Reasoning component only needs to be customized to provide the required subset. The following discusses two reasoning strategies.

3.2.1. Partial-order events. If a component X must be notified every time a piece of shared data is deleted within the COTS software then the Reasoning component has to continuously observe the COTS software to watch such pieces of data. If the interface of the COTS software externalizes such deletions then the Reasoning component can simply periodically check the externalized activity and call component X each time it occurs.

However, COTS software rarely externalizes activities as they are required by a system. In such cases, reasoning must instead check for the occurrence of certain internal or external events that indicate the non-externalized activities of interest. For example, if the COTS software does not externalize data that was deleted then, instrumentation can detect whenever the “delete” key on the keyboard was pressed. Of course, pressing “delete” on the keyboard only sometimes eliminates elements on the screen. For example, in Rational Rose pressing “delete” only deletes an element on the screen if the element was selected. Thus, the Reasoning component could be implemented to observe the selection of elements and keyboard events and only call component X if “delete” was pressed, and at least one element was selected beforehand and not deselected thereafter.

The above scenario is a simple example of reasoning with partial-order events where events are things like "delete pressed" or "element selected." Partial-order events can be applied on external stimuli or internal activities to reason about temporal occurrences (i.e., while in state X, if Y happens but Z did not occur then...). Various techniques are available for temporal reasoning (Luckham and Vera, 1995). We applied these techniques extensively in the context of Rational Rose and Matlab/Stateflow to infer internal activities. For example, we observed that a particular sequence of low-level interactions between Rational Rose and the operating system implies that the application is shutting down (low-level interactions provided through instrumentation). Since this information is relevant to other components, the reasoning component was instructed to notify other components whenever Rational Rose was shutting down.

3.2.2. Caching and partial comparison. COTS software often has redundant user interfaces that allow users to perform the same activities in different ways. For example, in Rational Rose a user may create a design element by dragging-and-dropping it from the toolbox; by selecting a design element from the menu; or by using a keyboard shortcut. Although the external, instrumentation-observable user stimuli differ, the resulting internal activities are the same. In cases where many choices exist on how to perform the same internal activities, it is often significantly easier to detect internal activities after they are completed rather than trying to predict the effects of observable stimuli. We have used cachingand partial comparison as a mechanism for inferring internal activities from the changes they produce in a COTS component's observable state and data.

A naive implementation of this approach for observing data changes is to cache data and then periodically compare the cached data with the COTS software's current data. This naive implementation works well if the amount of "relevant" data is small but becomes unscalable otherwise.

To demonstrate that a more sophisticated version of caching and comparison, based on partial caching and comparison, can scale to large amounts of data, consider again our Rational Rose example. Rational Rose maintains potentially large design models, which makes full caching and comparison infeasible. To limit the amount of caching and comparison, we use a combination of instrumentation and partial comparison. The rationale is as follows. We know that a user can make changes to data in Rational Rose only through the mouse or keyboard. We thus use instrumentation to observe mouse and keyboard events. We also know that only selected design elements in Rational Rose can be modified or deleted. We thus cache design data when it is selected and only compare the cached, selected data whenever mouse or keyboard events occur. Design data that is deselected remains in the cache but it is not compared any more. This limits the scope of caching and comparison—instrumentation detects when to cache/compare (e.g., mouse event) and the Rational Rose API determines what to cache/compare (e.g., Rose's API identifies which elements are selected at the time of a request). Note that previously selected data remains in the cache and its consistency with Rose is guaranteed until it is selected again.

We found that this "GUI-driven" partial caching and comparison approach—where mouse and keyboard instrumentation detect when to cache and compare and the COTS product's selection mechanism determines what to cache and compare—is applicable to a

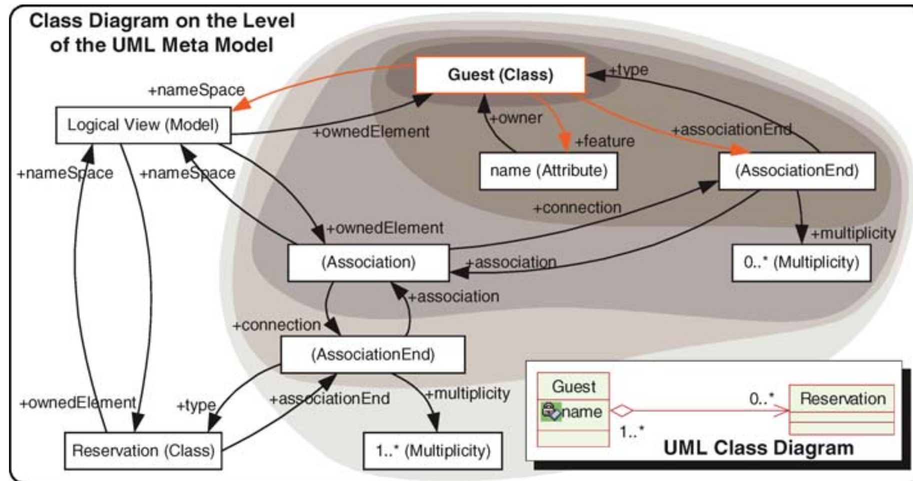

Figure 3. Propagating Change Detection. Example Class Diagram in bottom right. Meta Model of Example Diagram in center. Deletion of Class *Guest* triggers subsequent deletions that can be detected through caching and partial comparison.

variety of COTS software (e.g., Rational Rose, Matlab/Stateflow, Microsoft PowerPoint) with one additional caveat. Sometimes, COTS software may allow the modification of non-selected elements. Indeed Rational Rose has such exceptions. For example, when Rose deletes selected design data then it may also delete related design data that cannot exist without the deleted one. Figure 3 (lower right) shows a simple class diagram created in Rose. The diagram depicts two classes (Guest and Reservation) and a simple association relationship between them. If the class *Guest* is deleted then Rose also deletes dependent design elements that are related to the class, in this example the attribute *name* of the class *Guest* and the association relationship to it from *Reservation*.

Our *selection* based caching and comparison method will only detect the deletion of class *Guest* since it was the only selected element and consequently the only element that was cached and compared. There are two solutions to this problem. The first solution augments the Rational Rose method that returns selected model elements to also include the attributes and relationships of selected classes. This extended selection based caching and partial comparison will correctly detect the deletion of all involved design elements. This solution is simple since it approximates the notion of what elements are selected by assuming the worst-case scenario. Unfortunately, it raises scalability concerns because the assumed worst-case must always be compared for changes.

The second solution avoids the scalability concern by limiting comparision to only the truly selected design elements. If a selected design element has not changed then none of its related design elements can have changed either. Only the related design elements of changed selected elements must also be compared. For example, figure 3 (main part) depicts the Guest/Reservation class diagram example from the perspective of the UML meta model. It shows the model element *Guest* of type *Class*, its attribute *name*, and its

relationship to *Reservation* (note: UML association relationships have association ends that attach to classes). A deletion of class *Guest*, once detected through comparison, triggers a subsequent comparison of all other cached, related design elements. If an element did not change then no further comparison is necessary. If an element did change then its related elements (if any) also need to be compared. In the example in figure 3, caching and subsequent comparison is similar to a wave that originates at the selected element(s), navigates through all related elements, and terminates at those elements that have not changed. In this example, only two elements will remain after the wave of comparisons: *Logical View* and *Reservation*. This second solution is computationally much more efficient than the first solution since it limits the necessary comparison to a minimum.

Caching and partial comparison may be elaborate to implement but it can be very efficient in tracking changes to large data stores of COTS software. We used this technique successfully on Rational Rose, Matlab/Stateflow, and Microsoft PowerPoint with up to 37000 data elements (e.g., model size in Rose). These COTS software products had in common that they did not provide change notifications to data stores but they did provide access to selected elements upon request. In all three cases, caching and partial comparison was used to detect changes to data. See case study later for a concrete example.

3.2.3. Approximation. Although instrumentation and reasoning may sound simple in principle, it is often hard and elaborate to implement for a particular COTS product: Instrumentation often produces low-level, operating-system events; and reasoning may have to duplicate some of the functionality of the COTS software to infer relevant internal activities from external stimuli. Fortunately, instrumentation and reasoning is only required in as much detail as necessary to infer *relevant* internal activities. Also, the complexity of integration depends heavily on the accessibility of information relevant to infer internal activities. Even in cases where it is hard to infer internal activities precisely, we found that there are integration scenarios where it is good enough to approximate internal activities (i.e., false positives or negatives are acceptable).

As an example, let us reconsider the integration of Rational Rose and the consistency checking software discussed in the introduction. This software, called UML/Analyzer, checks the validity of UML diagrams (Egyed, 2001) created in Rational Rose. Due to the computational complexity of consistency checking, it was unrealistic and non-scalable for the UML/Analyzer to check the *entire* model frequently. However, validating only the consistency of changes as they occurred was feasible. We thus used our framework to observe changes to the Rose design models and to forward these changes to the UML/Analyzer for consistency checking. In this setting, false negatives (not reporting changes that happened) were not acceptable because the consistency checking software would fail to evaluate some changes. However, false positives (reporting changes that did not happen) were acceptable because re-evaluating a part of a model that was already validated is at most a waste in computation but not an error. Allowing false positives and/or negatives is a useful approximation and it can save computational effort and cost. That is, approximation algorithms typically execute faster and are cheaper to implement.

3.3. Mediation and translation

Returning to the COTS augmentation framework presented in figure 2, we now consider the use of mediation and translation to construct alternative interfaces for COTS products to facilitate their use as components in larger systems. Mediators and translators (Egyed et al., 2000) 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 (Williams and Kindel, 1994), CORBA (Object Management Group, 1995; Vinoski, 1997), DLL, RMI (Sun Microsystems, 2001)) 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 2). The services of the COTS software are then provided in the alternative format without other components and the COTS software being aware of this. An optional data store may be required if the augmented interface provides extended services.

The UML/Analyzer (Egyed, 2001) also serves as a good example for the need of an augmented interface to Rational Rose. Although Rational Rose supports the drawing of UML diagrams, its native interface does not describe those diagrams according to the UML standard. We thus created an alternative interface for Rose that conforms to the UML 1.3 standard. The alternative interface translates requests to and from Rational Rose. The UML/Analyzer software uses the alternative interface that behaves as expected by the system. This also has the advantage that the augmented COTS applications become more interchangeable. After we also built a UML interface for Matlab/Stateflow we were able to interchange Rational Rose with Matlab/Stateflow and vise-versa in several COTS-based systems (e.g., the simulator discussed later).

The alternative interface also mediates between two communication standards. Rational Rose provides its native interface through the Microsoft COM middleware (common object model, (Williams and Kindel, 1994)) whereas the UML/Analyzer, implemented in Java, prefers access to a Java UML library. The alternative interface was thus implemented in Java to "hides" the platform-dependent Microsoft COM middleware of Rose completely.

4. Rational rose augmentation infrastructure (RAI)

This section discusses a concrete example on how we augmented IBM Rational Rose. The two primary purposes of the augmentation were (1) to provide a UML-compliant interface for accessing Rational Rose data (UML model elements) and (2) to provide change notification whenever its UML design data was modified. The later also included change notifications in response to model loading or Rational Rose shutting down since it affected the availability of data. The Rational Rose Augmentation Infrastructure (RAI), depicted in figure 4, refines the schematic COTS integration architecture from figure 2. The RAI has an augmented programmatic interface, called the *Data Manager*, that

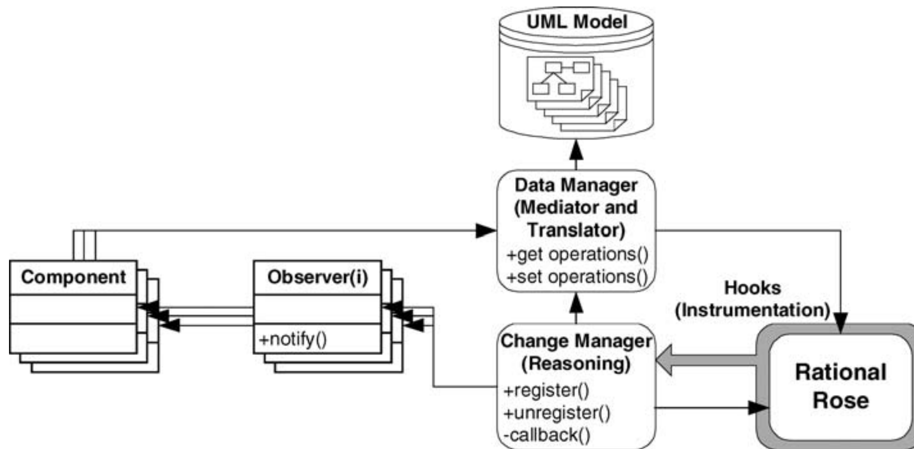

Figure 4. Rational Rose Augmentation Infrastructure (RAI).

implements the UML meta Model 1.3 completely. The Data Manager provides classes and methods that reflect the 150+ types of model elements in UML; in particular, it provides a class for every type of UML model element and it provides class methods for the attributes and relationships of those elements (over a thousand methods in total). The most basic use of the Data Manager is to create and maintain UML model elements, and to translate Rose design data into them. Other software components may use the provided, augmented classes and methods to create and maintain UML model elements in Rose. The Data Manager is the reactive part of the RAI as it responds to requests only (Egyed, 2002a).

The proactive part of the RAI is the *Change Manager.* The Change Manager acts as a broker between Rational Rose and the potentially large set of client components that may be interested in its activities. Figure 4 shows the use of observers that, coupled with the Change Manager, forward change notifications from IBM Rational Rose to other software components. Specifically, observers are registered to change managers to indicate interest in being notified about changes within the COTS software (see also observer design pattern (Gamma et al., 1994)). Observers are built to satisfy system-specific interaction needs of the components interested in COTS software's activities. The evaluation of what to notify and how to notify is left to the observers (i.e., filtering and syntactic/semantic transformations like data type conversions). If components reside on distributed nodes (e.g., different machines) then the observers also have to manage transportation issues (e.g., remote method invocations, (Sun (Microsystems, 2001)). The Change Manager utilizes caching and partial comparison as its primary means of detecting data changes within Rational Rose. This requires access to Rational Rose and the Data Manager. As was described earlier, some instrumentation was required to capture user stimuli (mouse and keyboard events) and other low-level activities that are associated with shutting down Rose.

With 30,000 lines of Java code, the size of the Data Manager is quite large because it implements the UML meta model completely. The Data Manager optimizes data caching

so that the same data is never translated twice from Rose to UML. The Change Manager uses roughly 1500 lines of Java code and is very efficient due to its caching and partial comparison approach. The instrumentation consumes less than 100 lines of C code. The RAI is lightweight and fast due to efficient instrumentation and reasoning. Even on very large models, its existence is mostly unnoticeable to human users. It was tested on over a dozen Rose models with up to 37,000 model elements.

The RAI encapsulates IBM Rational Rose through a well-defined and sound architectural framework. The use of a sound architectural framework in turn makes it easier to plug COTS software, such as Rose, into larger COTS-based systems. The framework thus improves the "plugability" of COTS software; the lack of which is generally seen as a significant reason for failures during component-based development (Boehm et al., 2000; Garlan et al., 1995). The next section will demonstrate the RAI on various integration scenarios that add simulation capabilities to Rose. We will show that using RAI is quite powerful since it enables the integrator to exert a great degree of control over the COTS-based system. Adding pro-active behavior to an otherwise reactive COTS software makes it possible to integrate this COTS software in ways that are impossible otherwise. Examples were discussed earlier. The next section will demonstrate this by discussing three different integration scenarios where the limits of integrating reactive COTS software are explored and the benefits of integrating proactive COTS software are presented. Later, Section 6 will summarize similar lessons learned while integrating other COTS software such as Mathwork's Matlab/Stateflow and Microsoft PowerPoint.

5. SDS simulator and rational rose case study

This section illustrates the use of the RAI for integrating the COTS software IBM Rational Rose with an in-house developed simulation software called *SDS Simulator* (Boehm and Basili, 2001). IBM Rational Rose provides a powerful modeling environment for UML-like designs and it is widely used. The SDS Simulator provides simulation capabilities for "executing" UML-like designs. The goal was to create a seamlessly integrated modeling and simulation environment that used Rose and the SDS Simulator as its components. A particular emphasis of the integration was on reliability and performance.

This section discusses three integration scenarios that were implemented and evaluated. The first two scenarios integrated the SDS Simulator with a reactive IBM Rational Rose. Both solutions have reliability and performance problems, and they did not satisfy all functional goals. The third and final integration scenario used instrumentation and reasoning to synchronize modeling and simulation. It satisfied all functional and quality goals.

5.1. Scenario 1: Simulator accessing data in reactive rose

The SDS Simulator required access to UML class and statechart diagrams. Since Rose did not provide an UML-compliant interface to access its class and statechart data, the SDS Simulator used the RAI Data Manager to translate Rose data into UML data. The first integration scenario, discussed in this section, integrated the SDS Simulator with Rational Rose using the RAI Data Manager only. Figure 5(a) shows this integration

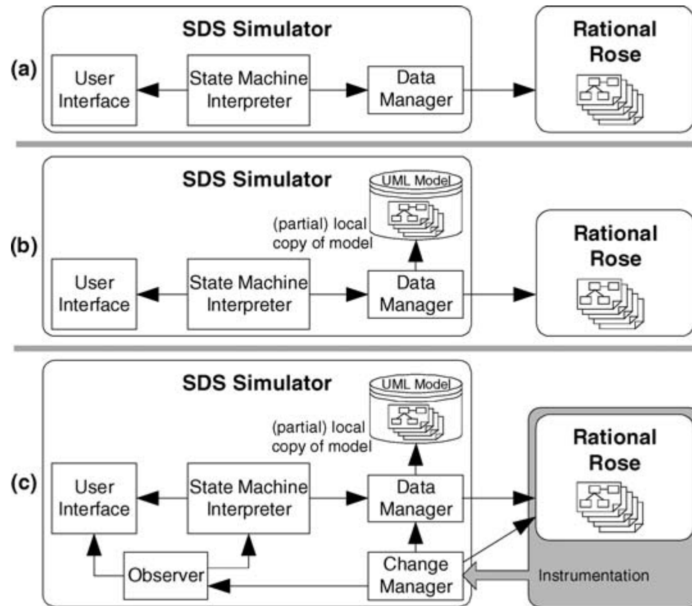

Figure 5. Reactive COTS Software Integration (a), (b); and Proactive COTS Software Integration (c).

scenario where UML models are created in Rose and accessed directly by the SDS Simulator through the RAI Data Manager.

Although this integration scenario was straightforward to implement, it was also very fragile and slow in performance. The performance of the simulator was slow mainly because of frequent, inter-process interactions between the simulator (the State Machine Interpreter) and the COTS software. The problem was that both, Rose and SDS Simulator, were executing in different processes and communicating through Microsoft's Common Object Model (COM). Inter-process COM calls are known to be computationally expensive because of marshalling activities. Since the simulator required frequent access to the modeling software (often to same or similar data) the interaction cost became significant. Independent of the performance problem and more severely was the reliability problem. Complex synchronicity issues were the result of both, Rose and SDS Simulator, executing concurrently and both having individual user interfaces. Synchronicity problems occurred whenever users made changes to the UML models in Rose while simulating them at the same time. For example, fixing a modeling defect during simulation caused such synchronicity problems because it resulted in inconsistencies between the simulator's internal data and Rose. In other words, the simulator was not aware of data changes in Rose. Abnormal and undesirable program exceptions were typically the response.

Although the RAI Data Manager itself was fast and reliable, the choice of architectural style on how to integrate Rational Rose and SDS Simulator through the Data Manager led to a fragile system with serious data synchronicity and performance problems. User caution was required to ensure proper functioning of the system. We encountered

similar integration problems with other COTS software like Microsoft PowerPoint and Matlab/Stateflow.

5.2. *Scenario 2: Simulator caching data of reactive rose*

The second integration scenario reduced the performance and reliability problems but did not eliminate them. This scenario also sacrificed some of the desirable seamless integration yet improved upon the performance by (partially) caching the data model of Rose before commencing simulation. Figure 5(b) depicts this somewhat better solution and shows that the SDS simulator maintained of its own, local copy of the Rose model (the UML model) which was downloaded, translated, and maintained by the RAI Data Manager. This solution was an improvement in terms of performance because accessing the model via the cached local copy (once established) was faster than the inter-process COM calls and translations with Rose (i.e., data accessed multiple times need not be downloaded and translated again). The RAI Data Manager supported both incremental caching and full caching, both of which had drawbacks in this integration scenario: caching takes time, it is still not (fully) reliable since model changes may happen during that time, and even minor changes require re-caching to simulate them. All these issues were undesirable but the last one also changed the functionality of the integrated system since the user needed to be aware of the current state of the modeling data in the simulator. New models could only then be simulated if the user told the simulator to download the latest version from Rational Rose. Advanced features, such as fixing defects during simulation, could not be implemented.

5.3. *Scenario 3: Simulator interacting with proactive rose*

While the second scenario improved performance and reliability somewhat, the remaining problems were architectural in nature and could not be improved upon by integrating a reactive COTS software. To provide better integration between the SDS Simulator and Rose, three major challenges had to be resolved:

- Prevent users from making changes to Rose while caching is in progress
- Update the local UML model whenever the Rose model changes
- Update the current simulation state (while running) whenever the Rose model changes

Architecturally, to resolve all these challenges, Rose had to become an proactive component in communicating changes of its model data to its neighbor component, the SDS Simulator. Instrumentation and reasoning enabled an architectural framework for doing exactly that (figure 5(c)). As was discussed in Section 4, instrumentation and reasoning can detect changes to data within Rose (by comparing selected parts of the Rose model with the UML model). Information about changes is then forwarded to registered observers. In case of the SDS Simulator, the role of the observer was to interpret changes within Rose to ensure synchronicity and consistency between it and the simulator. The SDS Simulator observer thus used change information (1) to update the local UML model and (2) to update the simulation state. For example, the deletion

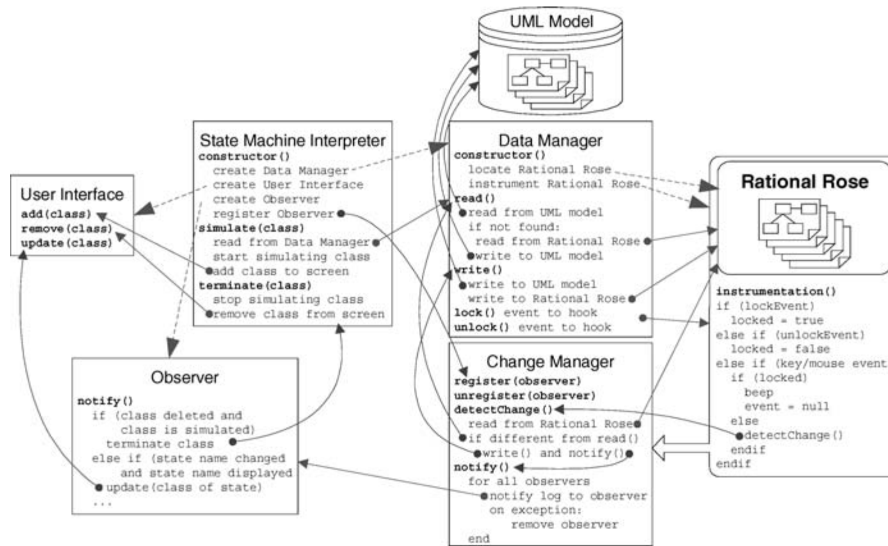

Figure 6. Pseudocode for Proactive SDS Simulator/Rational Rose Integration (this figure is a refinement of figure 5 (c)).

of a simulated object caused its termination if it was being simulated at that time; or the change of a state name caused the simulator to instantly update the new state name in its own display.

Figure 6 describes in more detail the interactions among Rose, its instrumentation and reasoning, and the SDS Simulator. Hooks (Balzer and Goldman, 1999) monitored operating system interactions with Rational Rose including mouse and keyboard events. It required very little code (roughly 100 lines of C code) to built hooks for Rational Rose to intercept keyboard and mouse events and to invoke the detectChange method in the Change Manager when these events occurred. The detectChange method, in turn, was responsible for reading selected design data from Rose and, if they differed from the cached design data in the UML model, updating the UML model. DetectChange also called notify to communicate the change to other components.

It was the responsibility of the observer to actually realize how to send change notifications to interested components; i.e., possibly filtering unimportant messages and performing syntactic and semantic transformations. In figure 6 the observer was a COM object and COM handled all communications. The observer called model synchronization methods of the simulator once the Change Manager notified it about data changes. Observers were not created automatically as part of a component's interface to Rose. Instead, only a template was provided for constructing such observers so that their interfaces conformed to our framework's architecture for interfacing with the Change Manager. In case of the SDS Simulator, updating the simulator's user interface was one of two synchronization tasks that the observer had to perform. The second task was to update the running simulation if it was currently active. For instance, guard conditions could be updated while the simulation unfolded. Changes could also occur that invalidated the simulation (i.e., the current state of simulated object is deleted). In such

cases, the observer either shut down the current simulation or proceeded with a defined recovery process. Augmenting proactive behavior to Rose solves two out of the three integration goals outlined earlier. It does not yet resolve the reliability problem. To satisfy integration reliability, it was imperative to prevent changes to design data in the Rose during certain critical periods (e.g., during caching). This capability was achieved by using instrumentation yet again (hooks) to actively block user input to the COTS software during these critical periods. Instrumentation simply issued a "beep" sound to let the user know that their input was being blocked temporarily.

The above discussion demonstrated that (1) we were able to convert a reactive COTS software product (IBM Rational Rose) into a proactive software component able to communicate with the simulator and that (2) it made a difference in the quality and functionality of the integration (e.g., avoided serious data consistency and state synchronicity problems). The contribution of our work is an infrastructure for augmenting the behavior of IBM Rational Rose (and other reactive COTS software).

6. Discussion

6.1. Cost effectiveness

Our approach showed that it is technically feasible to convert reactive COTS software into proactive ones. Of course, the cost and effort of doing so may be high. Here, a management decision is needed which depends on factors such as: (1) is there an alternative COTS software product that could be integrated more easily; (2) is it cheaper/less time consuming to re-built needed functionality in-house instead of using the COTS software; (3) is it possible to integrate the needed COTS software product reactively and still accomplish all integration goals; (4) is it possible to change (relax) integration goals to make it more cost effective. All these questions have nothing to do with the technical feasibility of using our approach and thus answering these questions is outside the scope of this paper.

It must be noted that in case of integration IBM Rational Rose and other COTS products it was clearly cost effective to use our approach. While our integration framework is large, it is small in comparison with the COTS products. We found that instrumentation tended to be simple, cheap, and computationally insignificant while reasoning tended to be complex and expensive to build (although still computationally efficient) (Boehm and Basili, 2001). In terms of development cost and effort, we found that reuse was clearly better and cheaper than re-development in all our case studies although we found it frustrating at times to work around COTS software limitations (Maiden and Ncube, 1998). There is a non-obvious trade-off between the cost of re-development and reuse. Given the diverse nature of COTS software there is no simple way of predicting which is better. Unfortunately, many potential integration problems are discovered late in the development. We had several cases of late requirements changes because of this. Existing literature explores this trade-off in more detail (Lawlis et al., 2001; Sedigh-Ali et al., 2001).

Although this paper only described the augmentation infrastructure for Rational Rose, it should be noted that only a few details of this infrastructure were specific to this COTS product. Over 95% of the code of the Rational Rose Augmentation Infrastructure was

identical with the equivalent one for Matlab/Stateflow. The differences were entirely in their instrumentation and translation code.

6.2. *Improved plugability*

Software components become exchangeable if their structure and behavior is consistent. The structure of a software product is defined by its programmatic interface. We discussed that the programmatic interface of COTS software can be augmented by implementing an alternative interface. For example, we implemented a Data Manager consistent with the RAI Data Manager for Matlab/Stateflow. Similarly, we discussed that the behavior of COTS software can be augmented also. For example, we also implemented a Change Manager consistent with the RAI Change Manager for Matlab/Stateflow.

Having a Data Manager and a Change Manager for Matlab/Stateflow consistent with the RAI made it possible to exchange IBM Rational Rose with Matlab/Stateflow arbitrarily. As such, the SDS simulator can use either IBM Rational Rose or Matlab/Stateflow as its graphical front-end with no changes to its code because both COTS software products have identical access and notification interfaces. This form of "plugability" makes it possible to replace COTS software in COTS-based systems with only minimal impact on other components. Our framework thus improves the plugability of COTS software because newly developed components, like the SDS Simulator, can be built under the assumption that it is being integrated with idealized COTS components. The simulator can thus be made insensitive to the particular choice of COTS product (Rational Rose, Matlab/Stateflow, or some other software) being used as its graphical front-end.

To date, we used the framework presented in this paper to integrate several large-scale COTS products (e.g., IBM Rational Rose, Matlab/Stateflow, and Microsoft PowerPoint) with a wide variety of other software components.

6.3. *Augmented notation and semantics*

This paper emphasized on augmenting COTS software access and notification without changing COTS software notation and semantics. However, especially in the context of UML, numerous examples exist (Hofmeister et al., 1999; Medvidovic et al., to appear) on how to augment the notation and semantics of UML by overloading the meaning of its existing model elements. In support of the MoBIES project (DARPA's Model-Based Integration of Embedded Systems), we were asked to provide an integration framework for COTS software commonly used in that community. The lack of integration was seen as a major deterrent to model-based development supported by multiple perspectives. The challenge to overcome was that the MoBIES program defined a language (notation and semantics) for Embedded System Component Modeling (ESCM) to describe a challenge problem provided by Boeing. Due to some similarity between the ESCM and UML, it was decided to model the ESCM in IBM Rational Rose to benefit from the generic modeling software. Differences between the ESCM and UML were modeled in Rose using some of its extensibility mechanisms (e.g., tagged values and stereotypes). Additionally, an ESCM access and notification mechanism was created, similar to the one discussed in this paper, to translate Rose UML elements into ESCM elements. This

case study showed that it was possible to change the "appearance" of a COTS notation and semantics without actually changing the COTS software itself. Although Rose believed it was modeling UML diagrams, other MoBIES software components integrated with Rose believed that it was modeling ESCM diagrams. Mediation, translation, instrumentation, and reasoning ensured a seamless integration of both perspectives by bridging their differences. The only limitations were that the augmented notation and semantics could not be less restrictive than the COTS notation and semantics. For example, Rational Rose does not allow circular class inheritances and this restriction could not be removed.

6.4. *Validation and limitations*

The integration framework presented in this paper has been applied to three major COTS products produced by different vendors. In addition to IBM's Rational Rose and Mathwork's Matlab/Stateflow, the technique was also applied to 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 (Taylor et al., 1996); it was integrated with an automated class diagrams abstraction software (Egyed, 2002), the SDS Simulator for executing UML-like class and statechart diagrams (Egyed and Wile, 2001) the Boeing/MoBIES Translator and Exporter for modeling embedded systems (Schulte, 2002), 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 (Goldman and Balzer, 1999) or the survey authoring system (Wile, 2001). The SDS Simulator was the only system that was integrated with all three COTS products at some point in its development. Table 1 lists a variety of COTS-based systems we built to date.

While it is out of the scope to discuss the systems in Table 1 in detail, it must be noted that most of them required proactive COTS software to satisfy integration goals. The SDS Simulator discussed in Section 4 is one such example. Only a few of the COTS-based systems we've built could be satisfied with a purely reactive COTS software product. For example, we built a Java code generator for UML that generated Java code out of UML class diagrams. Due to the simple batch usage of this software, a reactive Rose was sufficient. Nonetheless, even in this simple case study, instrumentation was necessary to block user input to prevent model changes during code generation.

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 tests. For example, large-scale models (data) with up to 37,000 model elements were used to test the performance of data synchronicity and forced concurrent access was used to test integration reliability.

Table 1. COTS Software integration case studies.

Rational Rose	UML/Analyzer (Egyed and Medvidovic, 2000)
	UML Class Abstraction (Egyed, 2002)
	SDS Simulator (Egyed and Wile, 2001)
	Boeing/MoBIES ACL Property Translator and XML Exporter (Schulte, 2002)
	UML Model Browser
	UML Code Generator
Mathwork's	SDS Simulator
Matlab/Stateflow	UML Model Browser
	Boeing/MoBIES Change Event Analyzer
Microsoft PowerPoint	Design Editor (Goldman and Balzer, 1999)
	Survey Authoring (Wile, 2001)
	SDS Simulator (early version) (Egyed and Wile, 2001)

To date, our infrastructure has been used by several companies (e.g., Boeing, Honeywell, and SoHaR) and universities (e.g., Carnegie Mellon University).

6.5. *Integration styles and architectures*

Our framework encapsulates COTS software. In a way, the framework and the COTS software together form a single software component to be used in any system although it is more than a component since it also includes interface and behavior for interacting with other components within the system. This work does not propose or suggest appropriate integration styles or architectures. However, the framework is based on an "access and notification style." This abstract style can be refined in one of many concrete integration styles or communication mechanisms. Thus, the architectural style that our framework is based upon does not significantly limit the reusability of our framework. For instance, architecture description languages (Medvidovic and Taylor, 2000) (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 (Luckham and Vera, 1995), shared memory, explicit data connectors (Medvidovic et al., 1999), middleware platforms (i.e., COM or CORBA) or other communication methods. This abundance of interaction methods implies many different architectural styles.

The role of our integration framework is to identify internal activities within COTS software but this framework can be extended to define any computation necessary to interact with other software components in response to these internal activities. This includes event passing, remote procedure calls, sockets, etc.

6.6. *Open issues*

Our work does not address the versioning problem that is inevitable with COTS products. New versions of COTS software may not be compatible with previous augmentations.

While we have so far only experienced minor, easily resolvable incompatibilities with new versions of COTS products, it is certainly possible that major incompatibilities could arise in future releases. This possibility would make it more difficult and resource consuming to upgrade those COTS products. This issue is out of the scope of this work.

Finally, while our integration framework might be applicable to any COTS software, we have focused solely on COTS software with user-driven, graphical interfaces (GUI). This has limited our experience with the broader set of COTS software integration somewhat. Future work will investigate this issue further.

7. Conclusion

Some requirements cannot be satisfied if a system's software components do not behave as intended. Augmenting COTS software with proactive behavior makes it technically feasible to change the behavior of COTS software. This paper presented an approach for augmenting COTS software. It is useful only if the COTS software is not sufficiently proactive and these deficiencies are not acceptable to achieve the functional and/or quality goals of integrating it into a larger software system.

Our approach uses instrumentation and reasoning to realize system-specific, proactive behavior for COTS software, and mediation and translation to implement alternative, system-specific interfaces for COTS software. While the augmentation of interfaces of COTS software is common practice today, it alone is not sufficient for building COTS-based systems. It is our observation that augmenting the behavior of COTS software is vital for systems where the COTS software itself should become an active component in the system; i.e., this is usually the case in COTS software with user interfaces (GUI).

COTS software reuse is only then practical if the cost of building the infrastructure (mediation, translation, instrumentation, reasoning, etc.) is significantly lower than the cost of implementing needed parts of the COTS software itself. We can confirm that in all our case studies it would have been significantly more expensive to implement COTS-compliant components rather than implementing the augmentation infrastructure. This is in part because reasoning only had to duplicate small parts of the COTS software functionality; other functionality was either not of interest or it could be accessed reasonably well through the native interfaces of the COTS software.

We have conducted over a dozen case studies to validate our approach. We found that our framework is most useful in cases where (1) the observable state and data of a COTS product changes in response to external stimuli (e.g., user input) and (2) changes are restricted to an observable subset at the time of the stimuli (e.g., selected data).

Acknowledgments

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

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

Abts, C., Boehm, B., and Bailey-Clark, E. 2001. COCOTS: A software cots-based system (cbs) cost model. In: *Proceedings of the ESCOM 2001,* pp. 1–8.

Balzer, R. and Goldman, N. 1999. Mediating connectors. In: *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems.* pp. 73–77.

Boehm, B., Port, D., Yang, Y., Bhuta, J., and Abts, C. 2002. Composable Process Elements for Developing COTS-Based Applications, Technical Report, University of Southern California, Los Angeles, USA.

Boehm, B.W. 1989. Verifying and validating software requirements and design specifications. In: *Software Risk Management,* B.W. Boehm, (editor) IEEE Computer Society Press, pp. 205–218.

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

Boehm, B. and Abts, C. 1999. COTS integration: Plug and pray? *IEEE Computer*, 32:135–138.

Boehm, B., Egyed, A., Port, D., Shah, A., Kwan, J., and Madachy, R. 1999. A stakeholder win-win approach to software engineering education. *Annals of Software Engineering*, 6:295–321.

Boehm, B.W. and Basili, V.R. 2001. COTS-based systems top 10 List. *IEEE Computer*, 34:91–93.

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The unified modeling language user guide*. Addison Wesley.

Brownsword, L., Oberndorf, P., and Sledge, C. 2000. Developing new processes for cots-based systems. *IEEE Software,* 48–55.

Dean, J. and Gravel, A. (editors). 2002. *COTS-Based Software Systems,* Springer Verlag.

Egyed, A. 2001. Automated consistency checking between diagrams—The viewintegra approach. In: *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE).*

Egyed, A. 2002a. The API of the UML Interface, Technical Report, Teknowledge Corporation.

Egyed, A. 2002b. Automated abstraction of class diagrams. *ACM Transaction on Software Engineering and Methodology (TOSEM)*, 11:449–491.

Egyed, A. and Medvidovic, N. 2000. A formal approach to heterogeneous software modeling. In: *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE),* Berlin, Germany, pp. 178–192.

Egyed, A., Medvidovic, N., and Gacek, C. 2000. A component-based perspective on software mismatch detection and resolution. *IEE Proceedings Software*, 147:225–236.

Egyed, A. and Wile, D. 2001. Statechart simulator for modeling architectural dynamics. In: *Proceedings of the 2nd Working International Conference on Software Architecture (WICSA).* Amsterdam, The Netherlands, pp. 87–96.

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

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

Goldman, N. and Balzer, R. 1999. The ISI visual editor generator. In: *Proceedings of the IEEE Symposium on Visual Languages.*

Hofmeister, C., Nord, R.L., and Soni, D. 1999. Describing software architecture with UML. In: *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1),* San Antonio, TX, pp. 145–159.

Lawlis, P.K., Mark, K.E., Thomas, D.A., and Courtheyn, T. 2001. A formal process for evaluating cots software products. *IEEE Computer*, 34:58–63.

Luckham, D.C. and Vera, J.J. 1995. An event-based architecture definition language. *IEEE Transactions on Software Engineering.*

Maiden, N.A. and Ncube, C. 1998. Acquiring COTS software selection requirements. *IEEE Software*, 15:46–56.

Medvidovic, N., Rosenblum, D.S., Robbins, J.E., and Redmiles, D.F. to apear. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology.*

Medvidovic, N., Rosenblum, D.S. and Taylor, R.N. 1999. A language and environment for architecture-based software development and evolution. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE'99),* pp. 44–53.

Medvidovic, N. and Taylor, R.N. 2000. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93.

Morisio, M., Seaman, C.B., Parra, A.T., Basili, V.R., Kraft, S.E., and Condon, S.E. 2000. Investigating and improving a COTS-based software development process. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE),* pp. 32–41.

Object Management Group. 1995. *The Common Object Request Broker: Architecture and Specification.*

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

Sedigh-Ali, S., Ghafoor, A., and Paul, R.A. 2001. ware engineering metrics for COTS-based systems. *IEEE Computer*, 34:44–50.

Sun Microsystems. 2001. *Java Remote Method Invocation—Distributed Computing for Java.* (UnPub)

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

Taylor, R.N., Medvidovic, N., Anderson, K.N., Whitehead, E.J. Jr., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. 1996. A component- and message-based architectural style for gui software, *IEEE Transactions on Software Engineering*, 22:390–406.

Vinoski, S. 1997. CORBA: Integrating diverse applications within distributed heterogeneous environments, *IEEE Communications Magazine.*

Wile, D. 2001. Supporting the DSL spectrum. Journal of computing and information technology. *Journal on Computing and Information Technology* 9:263–287.

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