Switch or Struggle: Risk Assessment for Late Integration of COTS Components

Sören Blom, Matthias Book, Volker Gruhn, Ralf Laue
Chair of Applied Telematics/e-Business, Dept. of Computer Science, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany; Phone: +49-341-97-32330, Fax: +49-341-97-32339
{blom, book, gruhn, laue}@ebus.informatik.uni-leipzig.de

Abstract

The domain requirements of software projects often seem so specialized to developers that their original design does not incorporate any commercial-off-the-shelf (COTS) components. However, if major implementation problems are encountered at a later stage in the project, the integration of a COTS component that promises to solve those problems may become a desirable alternative to struggling on with the original implementation. While a number of methods and criteria have already been proposed for requirements engineering, risk assessment and candidate selection of COTS components, they were developed for application in the initial phases of a project and thus do not take into account the much tighter time and design constraints imposed in a later project stage. To spark discussion on necessary adaptations of the established methods, this position paper uses the example of a concrete project to illustrate the characteristics of "switch or struggle" situations and proposes an initial set of risk factors to be considered at that time.

1. Motivation

As a corollary to common rules for component-based design, conventional wisdom suggests that commercial off-the-shelf (COTS) components should be selected early in a project in order to make sure one does not write redundant code, but maximizes the potential for re-use, prevents architectural mismatch and designs the interface for most efficient integration – otherwise, it seems, the risks associated with deferring these issues may seriously diminish the chances of a project's successful completion [5].

These arguments are certainly true for development projects that intend to rely on COTS components from the start, but our experience is that in practice, a considerably larger number of projects initially do not plan for COTS integration but traditional in-house software development. However, if serious implementation problems arise unexpectedly in such projects, COTS component integration may suddenly become an attractive alternative after all.

We believe that even in such situations, when the opportunity for early COTS consideration has long passed and the component design is already finalized, the late integration of COTS components may still be a viable option in order to improve or even save the whole project. Obviously, the risks and benefits of switching to a COTS component at this time vs. struggling on with the in-house implementation must be assessed very carefully. However, we believe that developers in such a "switch or struggle" situation cannot simply resort to established COTS requirements engineering [7], risk assessment [6] and candidate selection [3] methods, as the late project stage enforces much tighter constraints than those methods assume. In this position paper, we therefore propose a number of criteria that may serve as guidelines for developers at such a decision point, and would like to stimulate discussion on strategies for successful late COTS component integration.

In the following sections, we first relate our own experience with a "switch or struggle" situation (Sect. 2) to illustrate the nature of the problem we would like to address in this paper. Drawing from our observations, we then identify an initial set of risk factors that we believe should be considered before deciding about any late COTS component integration (Sect. 3). Finally, we suggest issues for discussion and directions for further research (Sect. 4).

2. Project Experience

2.1 Context and Setup

Our hypotheses are spurred by observations of an ongoing 17-month project where we are currently developing tools for the specification and control of complex navigation structures ("dialog flows") in web-based applications – namely, a so-called Dialog Flow Editor that enables developers to model dialog flows in the graphical Dialog Flow Notation (DFN) and generate XML-based specifications out of them, and a Dialog Control Framework that interprets

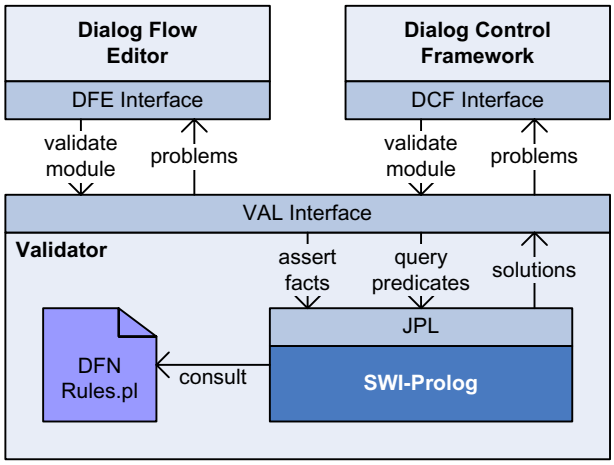

Figure 1. Validator with COTS Prolog prover.

those specifications and handles users' interactions with a web application accordingly [1].

Like any specification language, the DFN comprises a number of rules that define what makes a valid dialog flow specification. To make sure that the framework will not execute faulty dialog flow specifications that may lead to unpredictable application behavior, we must guarantee that the noation's rules hold both for the editor's diagrams and the framework's run-time interpretation of the generated XML specifications. To ensure that the exact same validation rules are applied by the editor and framework, we factored the validation logic out into a separate component (the so-called validator) that can be used by both tools (Fig. 1). Since the editor and framework work in different execution environments (a plug-in for the Eclipse IDE vs. a servlet-based framework deployed on an application server) with different data models (visual entities of a graphical editor vs. fragments of an XML-based Document Object Model), they are decoupled from the validator by interfaces that employ simple graph constructs to describe the dialog flows. The actual validation takes place in an off-the-shelf Prolog theorem prover, whose knowledge base is populated with facts about the dialog graphs and then queried for invalid constructs, using Prolog predicates for the DFN rules.

While this approach seems straightforward and natural in hindsight, it is quite different from the architecture we had first designed, since the decision to integrate the off-the-shelf Prolog prover was actually made very late in the project. In the following subsections, we will sketch the project's initial progress up to the point where we felt our original design was doing the validator more harm than good; then present the risks and benefits of the options we had to consider at this decision point; and finally relate our experiences after switching to the COTS component.

One key to understanding the design decisions discussed in the following sections is the organizational setup of our project: As a joint academic/industry endeavor, it is characterized by a higher degree of design freedom than a pure industry project - for example, concrete solutions for some challenges that are still a topic of current research (e.g. device independence [2]) were not yet completely worked out in the specification phase, but left to be addressed by the implementors of the respective components. Still, the project of course follows a strict schedule, which restricts the time developers can spend on solving those issues.

The editor and framework are developed in parallel by two essentially separate development teams under a joint project lead. While a prototype of the framework's core dialog control logic already existed as a starting point, the editor had to be built from scratch. The milestones in the project plans for both components are coordinated so the framework will at any time be able to interpret the XML specifications generated by the editor. Since neither the framework nor the editor contain any validation logic of their own, the validator was developed in a third parallel project "thread" during the first months of the editor's and framework's implementation. A common milestone for integrating the validator with the two "client" components was defined in the project plans for all three teams.

2.2 In-House Implementation Attempt

In the initial design phase (P_0 in Fig. 2), we had designed an interface for the validator component that abstracted from the editor's and framework's data model by representing dialog flows as simple directed graphs.

In the succeeding implementation phase (P_1), we began to develop Java algorithms that searched for violations of any DFN rules in the dialog graph. Unfortunately, after implementing the first few rules, it became apparent that the code fragments for checking individual rules were often distributed over several different methods, as certain properties had to be verified at different times of the graph traversal. Also, as a consequence of the fragmented implementation of individual rules, the node and edge classes originally designed as simple interface entities accumulated more and more state information, transforming them into a third dialog flow model in its own right. We realized that for these reasons, the further implementation of the validator would turn increasingly complicated and error-ridden.

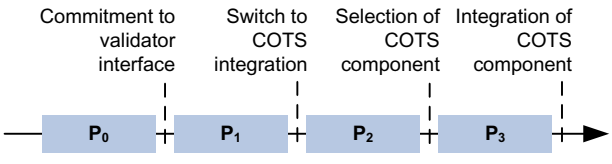

Figure 2. Component integration timeline.

At this point, the validator team took a step back to reassess their needs and wishes: Ideally, it should be possible to express and check the DFN rules individually and close to their mathematical form, as they are originally stated as invariants in predicate logic. Moreover, it should be possible to define subsets of rules that could optionally be skipped if they are irrelevant in a certain context or implicitly enforced by the editor and framework already (a capability would help in reducing the computational cost of validating a complete dialog flow specification). Most importantly, any alternative solution would have to comply with the interface that the other teams already relied on.

2.3 "Switch or Struggle" Decision

Thus, we had decide between struggling on with the original procedural implementation of the validator, or switching to a more declarative approach. We soon concluded that starting over and implementing a declarative validation logic from scratch was not feasible, since we lacked the time and skills to write our own theorem prover. Therefore, the only alternative was to find a COTS component capable of proving predicates.

Table 1 lists the factors influencing our decision most: Our expected progress in continuing the procedural implementation would probably be steady, but impeded by many refactoring and bug-fixing iterations. On the other hand, the progress in finding and integrating the right validation component would be less predictable, but promised a fast track to a purely domain-centric implementation and testing.

The expected integration effort would only be a challenge for the COTS alternative, since it depended on technical complexity of any glue code required. Preferably, the component would provide a Java interface, but even that would still have to be mapped to the interface of the validation component.

Regarding future maintainability, the in-house implementation would probably turn out messy, as modifying individual rules (whether for extensions to the DFN or just corrective maintenance) would require touching code in several places. In comparison, maintenance of individual rules expressed as predicates would be localized to single spots in the code.

Table 1. Expectations for in-house validator implementation vs. COTS prover integration.

	in-house implementation	**COTS integration**
progress	slow, but steady	hard to predict
integration effort	none	large, depending on component
maintenance effort	high	low
confidence in correctness	low	high

For these reasons, the overall confidence in a correct validator implementation that would be true to the specification was low for a continued in-house implementation, but quite high for reliance on the COTS component. However, while the COTS integration seemed to have clear benefits in the long run, it was still the more risky alternative. Nevertheless, we decided to switch to this approach since we already had an interface and internal data model for the validator. These could be "reinterpreted" as an enhanced connector that would comfortably hide the interface of any COTS component we would end up choosing, and allow us to make compromises regarding its technical details. On this basis, we deemed the risks of a COTS component's integration to be justifiable even this late in the project.

2.4 Component Selection and Integration

With this strategy for the technical integration worked out, our actual component selection process in phase P_2 could then focus much more on organizational aspects. Especially important to us were little or no additional expenses for licensing the COTS component, yet freedom to redistribute it as part of a commercial product; the possibility to declare DFN predicates in Prolog, since we had the necessary skills in-house; and proven maturity of the component and quality of its documentation.

After considering different candidates, we decided on SWI-Prolog [9] since it fulfills our requirements and comes with the bidirectional Prolog/Java interface JPL. The process of retrofitting our validator with SWI-Prolog (phase P_3) was largely accomplished by employing the validator's existing graph model and augmenting it with code to translate the properties of nodes and edges into Prolog facts. The JPL interface provides access to a prover instance that consults DFN rules formulated as Prolog predicates. Even while JPL has its quirks, the integration took place without major problems.

In order to validate a dialog graph, the editor or framework pass the respective node and edge objects to the validator component, which asserts the corresponding facts in the Prolog knowledge base and then evaluates the rule predicates, which yield solutions indicating any violating entities. These solutions are mapped back to the original node and edge objects, which the validator returns to the calling component with suitable error messages or warnings. The editor or framework can then react appropriately by displaying the messages and/or highlighting the offending graph elements.

2.5 Emerging Benefits

Since the project is still in progress, a final judgment of our decision's overall pay-off cannot be made yet. Still, we already see first benefits that we attribute to the integration of the COTS component.

First and foremost, we observe a shift of focus from low-level algorithmic technicalities to high-level domain concepts. The developers now spend much more time discussing the actual semantics of DFN rules rather than implementation details of Java-based graph traversal algorithms. This said, we still encounter occasional challenges in the precise formulation of DFN rules in Prolog, or need to resolve technical issues in the glue code connecting our original interface and the JPL API. However, the ratio of technical vs. conceptual concerns clearly improves towards conceptual.

This shift also emerges in testing the validator. Since much less infrastructural code is produced, our testing efforts focus much more immediately on the DFN rules and their "implementation" in Prolog predicates (we consider SWI-Prolog mature enough to assume that failing tests are caused by errors in our predicates, not errors in the prover). This way, by testing the validator we are actually testing the accuracy of our specification.

Looking ahead, we expect that due to this strong domain focus enabled by the new COTS-based validator implementation, we will be able to finish development of the component earlier and with higher confidence in its quality.

3. Characteristics of Late COTS Integration

3.1 Project Situation

While the late exchange of a partially completed in-house implementation against a COTS component worked well in our project, we do not recommend it in general. We believe that ideally, developers should design systems diligently, stay committed to their designs (whether COTS-based or not), and not introduce late changes lightly. However, our experience taught us that a project may not progress ideally, so component developers may find themselves confronted with unforeseen major technical or domain-specific challenges. In this case, they still have three options in order to build a (hopefully) working component after all:

(a) continue the component's implementation according to the original design, struggling to circumvent or accept its flaws

(b) scrap the component's implementation so far and try to implement it using a different approach

(c) scrap the component's implementation and switch to a COTS component serving the same purpose

Out of these options, we consider (a) and (b) to be related "struggle" strategies insofar as the responsibility for finding an alternative implementation continues to rest with the developer. In contrast, resorting to a COTS component qualifies option (c) as a separate strategy ("switch").

Of course, switching is not a sure-fire strategy: While it may effectively evade or resolve problems in the original implementation, it comes at the cost of introducing late major design changes. This always carries numerous risks, such as additional effort required for familiarizing developers with the new design, adapting the existing control flow and data model implementation to the new component interface, re-testing the integration and fixing side-effects and errors. These risks will likely not be restricted only to the changed component, but affect the rest of the system and thus endanger the overall project schedule and budget.

The successful evasion of this gloomy outlook in our project prompted us to question whether we were just lucky, or if there are actually certain factors that can mitigate the risks and make late COTS integration a viable alternative in "switch or struggle" situations. Developers could then use those factors to assess whether the risks associated with late COTS integration outweigh its benefits.

3.2 Risk Assessment

Before going into detail on the risk factors, we should clarify our understanding of late integration in order to put the following discussions into perspective: We define "late COTS component integration" as "retrofitting an already committed-to component design with a COTS component". The emphasis on the fixed component interface is important since the dependence of client components severely limits our design freedom in a "switch or struggle" situation. This distinguishes it from the ideal case of early COTS component requirements and risk analysis where the COTS component would not adapt to, but define that interface. In the case of late integration, however, we need to consider the following risk factors:

Level of interface abstraction. If we assume that the component's interface was originally designed with in-house implementation in mind, the most obvious risks stem from this interface's robustness against changes in the component's internal implementation (especially when those changes are as radical as exchanging it against a COTS component). Since changing an already published interface will require potentially far-reaching changes in the implementations of client components, we do not recommend switching to a COTS component late in the project if it would require changing the existing component interface.

However, a well-designed interface that abstracts from technical details of the client components and its own implementation should be robust enough to survive the internal switch to a COTS component without changes. Still, it will be necessary to bridge the gap between the original component's domain-specific interface and the technicalities of the COTS component's API. The effort and risk involved in this process depend not only on how wide the gap between the two interfaces is, but also if the necessary skills are available on the project team.

In our case, the differences between the editor's and framework's data model prompted an independent graph-based validator interface that made no assumptions about the actual implementation of the component, so we could retrofit it with the COTS Prolog prover without too much effort, and without affecting the editor or framework.

Divergence of architectural assumptions. Besides any gaps between the formal interface declarations, there may still be a mismatch in the architectural assumptions [4] or design intent [8] of the original and the COTS component. Often, these assumptions relate to the data structures (e.g. valid input range, possible output range, return values for invalid input, failed operations, empty results etc.) and/or processing protocols (e.g. thread safety, statefulness, transaction handling, caching strategies etc.). In contrast to traditional COTS integration products, however, we see a reversal of precedence in "switch or struggle" situations: Here, the assumptions are not determined by the COTS components, but by the existing project context, which is much harder to change at this time and therefore requires more manual mediation in the form of glue code.

In order for a COTS component to be successfully integrated behind an existing interface, it must work under the same assumptions that were made in the original component design. Minor differences in these assumptions (likely related to data structures, such as type conversions) can be ironed out by glue code that mediates between the original component interface and the COTS component. However, if major differences (especially in processing protocols) are present, compensating for them in glue code may cost too much implementation effort, require unavailable special skills, or force unreasonable restrictions upon the COTS and/or client components. We therefore conjecture that late COTS integration does not bear substantial risk only if the architectural assumptions of the original interface and the COTS component diverge just in minor, preferably data structure-related aspects.

In our case, the original design assumed that the dialog graph's nodes and edges would be stored by the validator component for re-use in future validation runs. Since this behavior could be emulated in the off-the-shelf Prolog prover by retaining the relevant facts in the knowledge base, the way in which the editor or framework use the validator component did not need to be changed.

The previously discussed factors provide some generic guidance on whether the late integration of a COTS component is advisable at all. However, the risk associated with such a decision also depends on the actual choice of COTS component: Even when the original component interface and its architectural assumptions are uncritical, designers may still decide against late integration of any desirable component candidates on the market if the individual characteristics of these components introduce too high risks.

Technical complexity and compatibility. One of these component-specific risk factors is the effort required for the technical integration of the COTS component: Dealing with a component's dependence on obscure libraries, reliance on certain platform versions, incompatibility with other components and similar issues can incur equal or more effort than the pure business-level integration, or even prove impossible to resolve. Performance considerations regarding response time and memory footprint may reveal further risks, which typically affect the whole system.

To minimize the risks of late COTS component integration, these factors should be assessed for each candidate component. While technical integration issues can usually be identified with reasonable effort using a prototyping approach, performance issues typically do not become apparent until the COTS component is already integrated into the system and working under production load. Usually, a thorough COTS component selection process would involve simulations that alert developers of such issues early, but in a "switch or struggle" situation, the time frame will likely be too tight to set up performance simulations for all candidate components. A pre-integration survey of other users' experiences with a particular component cannot eliminate these risks, but possibly give an indication of any problems to expect. We claim that the technology-related risks of late COTS integration can be kept within reasonable limits if such a prototyping and research phase precedes the actual choice and integration. If all candidate components seem to pose major integration challenges, we recommend that developers stick to their original implementation plans so as not to introduce unquantifiable risks into the project, even if the COTS integration would be desirable for other reasons.

In our case, the technical issues were the main focus of our decision process: Prototyping had shown that SWI-Prolog could be integrated into a Java environment using the JPL library. Although the technical integration seemed non-trivial (for example, it required the inclusion of native libraries on Windows and Linux platforms, as well as bundling the SWI-Prolog engine with the Prolog implementation of the DFN rules), we considered these challenges

manageable and thus proceeded to integrate SWI-Prolog.

Organizational constraints. Last but not least, the feasibility of late COTS component integration often depends on organizational constraints. Again, these can be best judged per candidate rather than in general. Some factors that may heavily influence the decision for or against an otherwise desirable candidate are its licensing model, the availability of source code, the quality of documentation, the product's age and maintenance frequency, etc. Typically, these criteria can be readily determined before integration and thus do not pose actual risks, but rather introduce constraints on the set of suitable candidates.

To support the technical challenges of late COTS integration discussed above, choosing a candidate with complete documentation is certainly advisable, and having the source code available may save an integration project in situations where the COTS component does not integrate as smoothly as expected. However, we would generally deem it too risky to integrate a COTS component late if it is already foreseeable that its source code will need to be adapted, unless the planned adaptation is so minor that its success can be proven in a cut-through prototype. The other organizational constraints depend largely on the individual project context, so no generic advice can be given on them. For example, when liability for the final product is an issue, much higher attention must be given to the COTS component's maturity, reliability and trustworthiness.

In our case, we required that the COTS component would not limit our options for choosing a licensing model for our own product. SWI-Prolog fulfilled this prerequisite and also made a good impression regarding documentation and maintenance record, thus confirming our choice.

4. Conclusions

Our goal here is to raise awareness of project situations that may call for late COTS component integration, to show that it is feasible under certain circumstances (even when the project was originally not planned as a COTS integration project), and to identify an initial set of risk factors to be considered in such a situation. Due to a project's far progress at this point, we found that the risk assessment has to take different factors and more severe constraints into account than in a project's initial phases. Therefore, we hypothesize that established processes and methodologies for COTS risk assessment and selection are not suitable for late COTS integration. We therefore propose that methods for late COTS component integration, and especially strategies for weighing risks vs. benefits in such "switch or struggle" scenarios, are an important topic for future research in the COTS community, and offer our initial set of risk factors as a basis for discussion.

5. Acknowledgments

The described joint project with itCampus GmbH in Leipzig is supported by a technology support grant from the European Regional Development Fund (ERDF) 2000-2006 and funds of the Free State of Saxony. The Chair of Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

References

[1] M. Book and V. Gruhn. Modeling web-based dialog flows for automatic dialog control. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 100–109. IEEE Computer Society, 2004.

[2] M. Book, V. Gruhn, and M. Lehmann. Automatic dialog mask generation for device-independent web applications. In *6th International Conference on Web Engineering (ICWE 2006)*, pages 209–216. ACM Press, 2006.

[3] A. Cechich, A. Requile-Romanczuk, J. Aguirre, and J. M. Luzuriaga. Trends on COTS component identification. In *5th International Conference on Commercial-off-the-Shelf (COTS)-Based Systems and Services (ICCBSS 2006)*, pages 90–99. IEEE Computer Society, 2006.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[5] R. J. Kohl. Requirements Engineering Changes for COTS-Intensive Systems. *IEEE Software*, 22(4):63–64, July/August 2005.

[6] J. Li, R. Conradi, O. P. N. Slyngstad, M. Torchiano, M. Morisio, and C. Bunse. Preliminary results from a state-of-the-practice survey on risk management in off-the-shelf component-based development. In *4th International Conference on Commercial-off-the-Shelf (COTS)-Based Systems and Services (ICCBSS 2005), LNCS 3412*, pages 278–288. Springer-Verlag, 2005.

[7] V. Perrone. A wish list for requirements engineering for COTS-based information systems. In *3rd International Conference on Commercial-off-the-Shelf (COTS)-Based Systems and Services (ICCBSS 2004), LNCS 2959*, pages 146–158. Springer-Verlag, 2004.

[8] D. E. Perry and P. S. Grisham. Architecture and design intent in component & COTS based systems. In *5th International Conference on Commercial-off-the-Shelf (COTS)-Based Systems and Services (ICCBSS 2006)*, pages 155–164. IEEE Computer Society, 2006.

[9] J. Wielemaker. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, Dec 2003. Katholieke Universiteit Leuven.