

A Classification of Software Components Incompatibilities for COTS Integration

Daniil Yakimovich[⊗]
dyak@cs.umd.edu

[⊗]Experimental Software Engineering
Group
Department of Computer Science
University of Maryland at College Park
A.V. Williams Building
College Park, MD 20742
USA

Guilherme H. Travassos^{⊗,•}
travassos@cs.umd.edu

[•]Computer Science and System
Engineering Department
COPPE
Federal University of Rio de Janeiro
C.P. 68511 - Ilha do Fundão
Rio de Janeiro – RJ – 21945-180
Brazil

Victor R. Basili^{⊗,⊖}
basili@cs.umd.edu

[⊖]Fraunhofer Center - Maryland
3115 Ag/Life Sciences/Surge Bldg.
University of Maryland
College Park MD 20742
301-405-4770

ABSTRACT

Integration of software components into a system can be hindered by incompatibilities between the components and system. To predict the possible incompatibilities and the ways to overcome them during the integration activities, a classification of incompatibilities can be useful for software developers. This can be especially crucial for COTS-based software development, where a software system is being built out of potentially highly heterogeneous software components. The resulting system can have a complicated architecture due to the diversified nature of its components (e.g., a message-based system with object-oriented and procedural sub-systems), and the architectural incompatibilities of the COTS products must be overcome. Moreover, the functionality of the COTS software products must be taken into account during COTS integration. In this paper we present a classification of incompatibilities based on the properties of local component interactions. We believe that this classification can capture possible problems about software component integration in heterogeneous software systems, including architectural and functional issues.

1. INTRODUCTION.

Commercial-off-the-shelf software is developed by a third party and intended to be part of a new software system [McDermid, Talbert 97]. Usage of COTS products is growing, because developers hope that it will increase their systems quality and reduce development time. However, COTS based development implies specific problems (such as selection, integration, maintenance, and security) whose solutions can be illustrated by answering the following questions:

- How *to select* the most suitable COTS product in the market?
- How *to integrate* the COTS product into the new system?
- How *to maintain* a system that has components developed outside?
- How *safe* a COTS software product is?

These are just a few problems. In this paper we are going to discuss COTS integration and its impact on COTS selection. The importance of discussing COTS selection and integration show up when considering that COTS products are developed to be generic, however, being integrated into a system, they are used in a specific context with certain dependencies. The existence of mismatches between the COTS product being integrated and the system is possible due to their

different architectural assumptions and functional constraints. These mismatches must be overcome during integration and they have to be identified even earlier. Thus, a classification of mismatches or incompatibilities can be useful for COTS selection and integration.

There are some publications exploring integration architectural issues. For instance, [Gacek et al. 95], [Shaw 95], [Shaw, Clements 96] identify and classify architectural mismatches and styles. [Abd-Allah, Boehm 96] and [Gacek 98] deal with heterogeneous architectures. This is especially important for COTS development because a COTS-based software system can be built out of potentially highly diversified software components, which can result in a heterogeneous architecture (e.g., a message-based system with object-oriented and procedural sub-systems) for the software system. However, not just architectural mismatches must be considered for integrating COTS, but also the required functionality, non-functional constraints, and software developers expertise level.

A COTS product can have gaps in required functionality, it can have incompatible interfaces, different architectural assumptions, and it can conflict with other system components. Selecting suitable COTS products for a project can require finding a trade-off between different mismatches depending on the organization's development capabilities. For example, if an organization has a strong expertise in a functional domain but little experience in coping with architectural problems it can consider acquiring COTS products with less required functionality but with few architectural mismatches. On the contrary, if an organization is more experienced in architectures than in the domain it should select COTS products with as much functionality as possible, although there can be considerable architectural problems. The right selection can minimize the integration effort.

Therefore in this work we propose a general classification of possible types of mismatches between COTS products and software systems, which includes architectural, functional, non-functional, and other issues. We present a classification of incompatibilities based on the properties of local component interactions. We believe that this classification captures possible problems about software component integration in heterogeneous software systems. We expect that the incompatibility classification can help to estimate the effort (cost) of the integration of the COTS products prior to deciding about using a specific one. By utilizing it, software developers can decide about a COTS product early in the software process, anticipating the possible integration risks.

This paper has four sections including this introduction. Section 2 deals with the interactions and how such concepts can be explored to identify incompatibilities. The third section explores the whole model, showing which types of incompatibilities software developers should look for. Also, a short example of using such a scheme is presented. Section 4 concludes this discussion and shows some on going works regarding estimation of cost for COTS integration.

2. INTER-COMPONENT INTERACTIONS AND CLASSIFICATION.

The incompatibilities, for the context of this work, are essentially failures of components' interactions, so finding and classifying these interactions will help to find and classify the incompatibilities. We consider three aspects of inter-component interactions and incompatibilities: type of interacting component, layer (syntax or semantic-pragmatic), and number of components participating in the interaction.

First, the components interact with other system components, and with the system environment. System components can be either software or hardware (excluding everything related to the environment, such as CPU and memory, but including devices directly controlled by the system, such as on-board devices) that are used by the software system. The environment can be of the development phase, which includes compilers, debuggers, and other development tools, or it can be the environment of the target system, which includes Operating Systems, virtual machines (such as Java), interpreters (such as Basic), and other applications and utilities used by the target system. The parts of both environments can also be considered components. Figure 1 shows the different perspectives that can be used to classify these software component interactions.

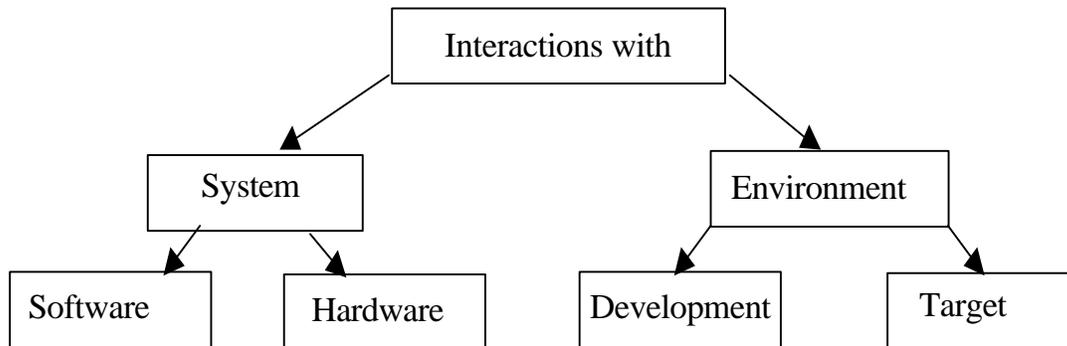


Figure 1. Interactions of software components.

Then two main layers can be differentiated in the inter-component interactions:

- **Syntax**, defines the representation of the syntax rules of the interaction, e.g., the name of invoked function; the names, types, and the order of the parameters or data fields in the message, etc. For instance, `float SQRT(float x)` represents a C notation for a function called “SQRT” returning a real result and with one argument, a real number `x`.
- **Semantic-pragmatic**, defines the functional (semantic and pragmatic) specifications of the interaction, i.e., what functionality is performed by the component, e.g., invoking the function "SQRT" calculates the square root of its only argument and returns it to the caller. However, in this work we do not consider semantic and pragmatic issues separately.

Finally, an incompatibility can occur in an interaction involving a certain number of participating components. A syntax incompatibility can occur because of syntactic difference between two components, but a semantic-pragmatic incompatibility can be caused either by just one component, two mismatching components, or three or more conflicting components. Thus, incompatibilities of the semantic-pragmatic layer can be classified according to the exact number of components that caused the interaction to fail. Therefore, the following types of semantic-pragmatic incompatibilities can be considered:

- **1-order semantic-pragmatic incompatibility**, or an **internal problem**, if a component alone has an incompatibility disregarding the components it is interacting with. It means that the component either does not have required functionality (not matching the requirements) or its invocation can cause a failure (an internal fault).

- **2-order semantic-pragmatic incompatibility**, or a **mismatch**, if an incompatibility is caused by interaction of two components. Both components may not have 1-order incompatibilities and can work correctly in other contexts. For example, a procedure that calculates the square root of a real number receives a negative argument from a caller that supposes that this is a valid output.
- **N-order semantic-pragmatic incompatibility**, or a **conflict**, if an incompatibility is caused by interactions of several components. There may not be semantic-pragmatic 1-order and 2-order incompatibilities for these components, but their cumulative interaction can cause a failure. For example, several processes together require more memory than the available amount, although each of them can be satisfied independently, so there is an n-order incompatibility on the semantic-pragmatic layer in interactions with the target platform.

According to the assumptions above, syntactical and semantic-pragmatic incompatibilities can occur in the system and environment dimensions. Table 1 captures this classification, where the cells are described below.

<i>Type of component</i>	<i>System</i>		<i>Environment</i>	
Type of incompatibility	Software	Hardware	Development	Target
Syntax	1.1	2.1	3.1	4.1
Semantic-pragmatic 1-order	1.2a	2.2a	3.2a	4.2a
Semantic-pragmatic 2-order	1.2b	2.2b	3.2b	4.2b
Semantic-pragmatic n-order	1.2c	2.2c	3.2c	4.2c

Table 1. Interactions incompatibilities.

1. Interactions with software

1.1. Syntax:

Three different types of syntax incompatibilities can be described here. Although there is only one cell capturing the idea of syntax issue for software in Table 1, its contents allows the identification of differences/incompatibilities regarding:

- Information flow, e.g., control instead of data.
- Binding: static, dynamic compile-time, dynamic run-time, topological, etc. As the result a component can not find another one.
- Interface protocol: different number of parameters or data fields, or different types of parameters or data fields.

1.2. Semantic-pragmatic:

1.2.a. 1-order: internal problem. These incompatibilities appear when the COTS product does not match the required functionality (e.g. it does not perform a required function), or due to its poor quality it still does not work properly (an internal fault). On the other hand, it can be other software that is solely responsible for the failure of interaction with the COTS product.

1.2.b. 2-order: different assumptions between two components, including the synchronization issue. These incompatibilities are products of a mismatch between the COTS product and other components surrounding it. Even when two components have correct functionality they can fail to work together due to some differences. (e.g., one object uses metric units, but another one uses inches, therefore the result can hardly be correct; another example is a mismatch between an asynchronous and a synchronous component).

1.2.c. N-order: a conflict between several software components. Even when the COTS product works correctly itself and correctly interacts with other components, some incompatibilities can appear as the result of a combined interaction with several other software components. (e.g., an object that controls rotation of a spacecraft receives the command for rotating on n degrees from a commanding object, but occasionally there is another commanding object, which sends the same command at the same time, in the system. Every single interaction is correct, but the spacecraft rotates twice as fast as it should do.)

2. Interactions with hardware

2.1. Syntax:

Different type of protocol. A software component can not work with a piece of hardware, because they assume different protocols (e.g. TCP/IP and Decnet or different port numbers).

2.2. Semantic-pragmatic

2.2.a. 1-order: wrong functionality of hardware or the COTS component. A hardware component does not work correctly (e.g. a printer does not support the Cyrillic alphabet), or the COTS component causes a failure.

2.2.b. 2-order: different assumptions between software and hardware. An interaction between software and hardware components does not work correctly (e.g., a program tries to print a Cyrillic text, but the printer has a different coding for the Cyrillic alphabet, therefore the output will be unintelligible).

2.2.c. N-order: a conflict between several software components over hardware. An interaction among several software components and a hardware component does not work correctly (e.g., several applications simultaneously accessing a single printer).

3. Interactions with the Development Environment

3.1. Syntax:

Different components' representation. The environment does not understand the packaging of a software component (e.g., a C program can not be compiled by a Fortran compiler).

3.2. Semantic-pragmatic:

3.2.a. 1-order: wrong functionality of the environment or the COTS component. The environment does not work properly (e.g., a defect in the compiler version), or the component has an error (e.g., a program can not be compiled because of a syntax error in it).

3.2.b. 2-order: different assumptions between the software component and the environment. A software component can not interact with the environment (e.g., a program is written in an old dialect of the language and can not be compiled by a newer compiler).

3.2.c. N-order: a conflict between several software components over the environment. An interaction among several software components and the development environment causes an incompatibility (e.g. two or more C modules can not be compiled or linked together because of a name collision).

4. Interactions with the target environment

4.1. Syntax:

Platform type. The environment does not understand the packaging of a software

component (e.g., a program uses another OS, or an interpreter can not run a program written in another language).

4.2. Semantic-pragmatic:

4.2.a. 1-order: wrong functionality of the environment or the COTS component. The environment does not work properly (e.g., the OS crashes), or the component has an error (e.g., a memory violation in a program).

4.2.b. 2-order: different assumptions between the software component and the environment. A software component does not interact with the environment correctly (e.g., a different version of the OS version performs some functions used by the component in a way other than expected by the component's developers).

4.2.c. n-order: a conflict between software components over the environment, including the control issue. An interaction among several software components and the environment causes an incompatibility (e.g. a conflict between two object-oriented frameworks in a one-process program for the control flow [Sparks et al. 96]).

3. TYPES OF INTEGRATION PROBLEMS.

Different incompatibilities have different solutions, but generally we can find five groups of related problems with the proper solution strategies. We assume that one type of incompatibilities can cause problems in different groups. For example, a syntax software incompatibility can cause different types of binding, which can require a special architectural solution for the whole system, or it can be just a different order of parameters, which can be overcome by a simple wrapper. Thus, we can differentiate the following groups of integration problems:

- **Functional.** All the 1-order semantic-pragmatic incompatibilities that are caused by missing or wrong functionality. Re-implementation or modification of faulty components can solve these problems.
- **Non-functional.** Some 1-order semantic-pragmatic incompatibilities can be caused by not matching to non-functional requirements, such as reliability, maintainability, efficiency, usability, etc. These problems are difficult to solve without reworking the component.
- **Architectural.** These issues constitute another class of problems and can cause changing the overall system's architecture, but the incompatibilities causing them are different. In this work we consider the following architectural assumptions of software components with their respective incompatibilities: packaging (syntax development and target environments), control (n-order semantic-pragmatic target environment), information flow (syntax software), binding (syntax software), synchronization (2-order semantic-pragmatic software) [Shaw 95], [Yakimovich et al. 99].
- **Conflicts.** Problems of this type are conflicts between components in the system (e.g., deadlocks). The related incompatibilities are n-order semantic-pragmatic software and hardware. The possible solutions can include changing the system's configuration without changing the overall architectural type (minor architectural changes, including monitoring components) and using glueware.
- **Interface.** These problems are incompatible interfaces between the components caused by some syntax and 2-order semantic-pragmatic software and hardware incompatibilities (other

than major architectural). The possible solution is glueware.

Another property of this high-level classification is that the classes of problems are specific to the particular development phases. Functional and non-functional issues require information on the project and COTS product functionality, which is available early in the requirements analysis phase. Architectural issues are dealt with during the design phase when the system's architecture is being designed. Conflicts and interface issues are addressed later in the design phase when the system's architecture and the component's interfaces are known.

Let us consider the following example to illustrate our approach; a 3D-graphics engine is being chosen for a real-time system. The system being developed imposes the following high-level requirements for the graphics engine:

Functionality: drawing 3-dimensional objects, including input and output 3D images from files.

Non-functional issues (portability): Mac.

Architectural issues (development platform): Ada 95.

Interfaces (example of a function): procedure Rect(x, y, w, h: Real); where (x,y) – the coordinates of the left bottom corner of the rectangle; w – its width; h – its height; output – drawing a rectangle. Other specifications, such as non-functional requirements, hardware requirements, possible conflicts, etc., are not considered in this example.

The possible candidate COTS products are OpenGL, QuickDraw3D, and DirectX [Thompson 96]. Matching them against the requirements gives the following data:

OpenGL:

Functionality: the drawing functions are provided, input and output from files is not supported – 1-order semantic-pragmatic incompatibility.

Non-functional issues (portability): Mac platform is supported.

Architectural issues (development platform): an Ada implementation is available.

Interface: procedure glRectf(x1:GLfloat; y1:GLfloat; x2:GLfloat; y2:GLfloat); where (x1,y1) – the coordinates of one vertex of the rectangle; (x2,y2) – the coordinates of the opposite vertex of the rectangle. There are a syntax incompatibility (different procedure names) and a 2-order semantic-pragmatic incompatibility (different interpretations of the arguments) with software components.

QuickDraw3D:

Functionality: drawing provided, input and output from files is supported.

Non-functional issues (portability): Mac platform is supported.

Architectural issues (packaging): Ada 95 implementation is not available – 2-order semantic-pragmatic incompatibility with the development platform.

Interface: it is not necessary to consider it, because it is expensive to use QuickDraw3D due to the different packaging.

DirectX:

Functionality: drawing provided, input and output from files is supported.

Non-functional issues (portability): Mac platform is not supported – 2-order semantic-pragmatic incompatibility with the target platform.

Architectural issues (packaging): Ada 95 implementation is not available – 2-order semantic-pragmatic incompatibility with the development platform.

Interface: it is not necessary to consider it, because it is extremely expensive to use DirectX due to the different packaging and target platform.

The result of this comparison is that OpenGL is the best candidate, despite certain incompatibilities that can be overcome using glueware and re-implementation. Use of C-implemented QuickDraw3D would require changing the system's architecture. Use of DirectX would require porting it to Mac, which is hardly a real operation.

4. CONCLUSIONS AND ON-GOING WORKS.

In this paper we presented a classification of incompatibilities between software (including COTS) components and other parts of a software system. This classification is intended to find the possible problems, including functional, architectural, non-functional, conflict, and interface, when a COTS software component is being integrated into a system. We hope that the incompatibility classification and the effort estimation approach can be useful for software developers to evaluate and integrate COTS software.

We have given above a classification of possible incompatibilities between the software (COTS) and other system components. However, to select a COTS product, developers must also know the effort required for overcoming these incompatibilities. To estimate the integration effort developers have to answer the following sequence of questions:

- *What are the incompatibilities?* - What is the difference between the system's requirements and the COTS products. This difference can be found using approaches, such as the comprehensive reuse model [Basili, Rombach 91].

- *How are they to be overcome?* - What integration strategies can be used by the developers to integrate the COTS software products (e.g., re-implementation, glueware, changes of architecture).

- *What is the amount of integration work?* - This is a quantitative estimation of the two items above; how much work is to be done to fill a certain gap.

- *What is the productivity (skill) of the developers for the applied integration strategy?* - This reflects the skill of the developers with respect to particular integration tasks. The higher it is, the faster they can perform the same amount of work. It can be possible to define techniques in different strategies, for example, re-implementation using object-oriented, procedural, or another paradigm. Specifying techniques within the strategies will demand more data about the organization, but on the other hand, the analysis will be more fine-tuned.

- *What is the effort required for overcoming a particular incompatibility between a COTS product and the system?* – This is obtained from the previous two items by dividing the amount of work by the productivity.

- *What is the total effort required for integrating a COTS product?* – This is the sum of the efforts required for resolving all the incompatibilities between the COTS product and the system.

Essentially, this is a bottom-up effort estimation model: each of the COTS product components is analyzed with respect to all its possible interactions with system to be integrated in. If an incompatibility is found the effort to overcome is estimated based on the amount of integration work and the productivity of organization for this type of work. The overall integration cost is the sum of overcoming all the incompatibilities between the COTS product's components and the system. However, to develop this COTS evaluation approach we must find effective ways to measure the productivity and the gap between the requirements and the system being developed.

As a research work, a process model for COTS selection, evaluation, and integration is being defined incorporating the ideas showed in this paper. Some experiments have been planned to empirically validate such a model. The results of these experiments, and the whole model, will be described in future publications.

REFERENCES:

- [Abd-Allah, Boehm 96] Abd-Allah, A., Boehm, B., "Models for composing heterogeneous software architectures", USC Technical report: USC-CSE-96-505, University of South California, Los Angeles, August 1996.
- [Basili, Rombach 91] Basili, V., Rombach, H., "Support for comprehensive reuse", *Software Engineering Journal*, September 1991, pp. 303-316.
- [Gacek 98] Gacek, C., "Detecting Architectural Mismatches During Systems Composition," Doctoral Dissertation, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089, December 1998.
- [Gacek et al. 95] Gacek, C., Abd-Allah, A., Clark, B., Boehm, B., "On the definition of software system architecture", in the proceedings of the First International Workshop on architectures for software systems – in cooperation with the 17th international conference on software engineering, Seattle, WA, 24-25 April 1995, pp. 85-95.
- [Garlan et al. 95] Garlan, D., Allen, R., Ockerbloom, J., "Architectural Mismatch or Why it's hard to build systems out of existing parts", *Proceedings of International Conference on Software Engineering*, 1995, Seattle, WA, USA, pp. 179 – 185.
- [McDermid, Talbert, 97] McDermid, J., Talbert, N., "The Cost of COTS" (interview), *Computer*, June 1997, pp. 46-52.
- [Shaw 95] Shaw, M., *Architectural Issues in Software Reuse: It's Not Just the Functionality, It's Packaging*, *Proceedings of the Symposium on Software Reusability*, 1995, Seattle, WA, USA, pp. 3-6.
- [Shaw, Clements, 96] Shaw, M., Clements, P., "A field guide to boxology: preliminary classification of architectural styles for software systems", http://www.cs.cmu.edu/afs/cs.cmu.edu/project/vit/www/paper_abstracts/Boxology.html, Computer Science Department and Software Engineering Institute, Carnegie Mellon University, 1996.
- [Sparks et al. 96] Sparks, S., Benner, K., Faris, C., "Managing Object-Oriented Framework Reuse", *IEEE Computer*, September 1996, pp. 52-61.
- [Thompson 96] Thompson, T., "Must-See 3-D Engines", *Byte*, June 1996, pp. 137-144.
- [Yakimovich et al. 99] Yakimovich, D., Bieman, J.M., Basili, V.R., "Software architecture classification for estimating the cost of COTS integration", *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, USA, 1999, pp. 296 –302.