Dealing with Software Components Compatibility

Alberto SILLITTI, Giampiero GRANATELLA, Paolo PREDONZANI, Tullio VERNAZZA
DIST – Università di Genova
Via Opera Pia 13, I-16145 Genova, Italy

and

Giancarlo Succi
Libera Università di Bozano
Piazza Domenicani 3, I-39100 Bolzano, Italy

ABSTRACT

Component Based Software Engineering (CBSE) is a development technique based on a wide use of components-of-the-shelf (COTS). Commercial components repositories contains hundred thousands components, that makes component selection an extremely difficult and time expensive task. Often, component selected by functional features are incompatible or the integration effort required is too high. Adding a selection of components based also on compatibility can simplify the integration task. This paper presents software components compatibility issues analyzed in the European founded project CLARiFi (CLear And Reliable Information for Integration).

Keywords: Component Based Software Engineering, Compatibility

1. INTRODUCTION

Component Based Software Engineering (CBSE) [1] considers components as black boxes that carry out some functionality but hide how they implement the functionality. Interfaces abstract component’s functionalities and work as the formal “contracts” used to build interconnections among components. An interesting formalisation of the interconnection problem is to consider components as black boxes exposing ports, connected through links [2]. Ports schematise the component’s interface and the way to access its functionality. Ports are connection points at a component’s boundary. Links are the association among the components’ ports. In the literature [2] [3] [4], the notations used to describe such ports are generally called Module Interconnection Languages (MIL) or Interface Definition Languages (IDL). The purpose of ports and links is to enable components to co-operate to produce a desired joint functionality.

Components that achieve this purpose are compatible. However, not all components are compatible. The problem we address in this paper is how to make potential compatibility/incompatibility evident to an integrator.

The paper addresses the compatibility problem using a notation based on stacks and layers. A stack is an abstraction (Fig. 1) representing the path of data – or messages – as they pass in and out of a component. In typical situations, there is a specialisation of the layers depending on their position in the stack. The upper layers represent the data formats, i.e., the structure of the information. In some cases, when the information meaning is detailed and fixed, upper layers may convey also the semantics of the data. Examples of data formats are PDF, domain-specific DTDs, JPEG, etc. On the other hand, the bottom layers represent “channels” of communication, such as protocols and APIs. Examples of channels are Java RMI and the POSIX file system API.

Fig. 1: Structure of a compatibility stack

Fig. 2 depicts a stack made of four layers. MyDTD specifies the semantically aware format of the data. XML provides a general format for MyDTD data. WAP is the application protocol used to send the content. SMS is the transport protocol used to send content.
This paper is organized as follows: section 2 and subsections analyze compatibility patterns between components; section 3 presents compatibility check between many components; section 4 discusses incompatibility issues; finally, section 5 draws the conclusions.

2. ROLES AND COMPATIBILITY PATTERNS

A “role” represents the possibility for components involved in an interaction to participate to the interaction performing different, complementary responsibilities. The example in Fig. 3 shows an MVC interaction pattern where three components play the roles of model, view, and controller.

Roles are a natural concept in patterns, architectures, architectural styles, reference architectures, etc. A consolidated terminology does not exist, although there are attempts of unification in the literature [5]. In this paper, we will use consistently the term “compatibility pattern” to refer to a pattern where two or more components interact.

It is responsibility of the compatibility check mechanisms to determine which components can actually interact, based on the patterns their respective suppliers have specified. In the following subsections, we will identify four compatibility patterns, implemented in the CLARIFi system, that are recurrent in practical cases. The approach is to identify few, easy to understand patterns to handle the majority of cases. This approach tends to simplify the theory and to justify it though practical applicability.

“Request-response” compatibility pattern

This compatibility pattern derives from client-server architectures and it is probably the most common (or, at least, new standards such as SOAP [6] show that there is a large interest in it). The pattern identifies two port roles: “client” and “server”. The client initiates an interaction by sending a request to the server; the server reacts by producing a response to the client (Fig. 4).

Since components can have several ports, it can be in the client role in some interactions and in the server role in others.

The request/response pattern requires ports to be compatible for both the request and response involved in the interaction. The ports need to have the same layers for the communication channel (e.g. RPC, RMI, HTTP, and SOAP) and for the data format (depending on the application).

Fig. 5 shows a possible port based on SOAP over HTTP communication, where the request is in DOC format and the response is in PDF format. The request and response stacks, however, are different (or split) in the upper part.

Within a single port, different stacks for request and response are possible. This introduces some complexity in the stack formalism. However, given the relevance in
many real cases, the advantages of the complexity outbalance its disadvantages.
As discussed, request and response stacks can share some common layers (e.g. the channels). Two representations are possible for this situation: one depicts the common layers as visually “shared”; the other presents two distinct stacks with duplicated common layers (Fig. 6). The former better highlights the common layers; the latter may require a simpler underlying data model. Anyway, the information content of the two representations is the same.

![Fig. 6: Equivalent representations for request/response stacks](image)

Conceptually, request and response stacks may have radical differences, such as a different number of layers. Fig. 7 depicts a possible port of the Oracle DBMS: the request stack shows a higher level of detail than the response one.

![Fig. 7: Different number of layers in request/response stacks](image)

“Shared data” compatibility pattern
Data sharing is a popular technique for component interaction. UNIX, for instance, uses shared memory for inter-process communication (IPC). Technologies that are more recent allow data sharing in distributed environments: JavaSpace, for instance, allows Java component to access a space of shared objects.
The “shared data” compatibility pattern assumes that components behave symmetrically in the interaction: hence, the only supported role is “peer”. The pattern allows read, write and concurrency control (e.g. locks) operations, with details depending on the specific technology.
Coherently with the stack theory, ports include layers for the channel (e.g. the technology, protocol, or API that allows sharing) and for the format of the shared data. The data format can be:

- Specific to the data sharing technology – e.g. Java (Fig. 8a)
- Technology independent – e.g. XML (Fig. 8b)
This compatibility pattern requires a single stack per port (as opposed to the request-response pattern that requires two).

![Fig. 8: Examples of data sharing compatibility pattern](image)

“Pipe” compatibility pattern
Piping of input/output between components is a popular composition technique in UNIX shell commands, in digital signal processing, and generally in applications that can be modeled using a data-flow paradigm.
The compatibility pattern comprises two roles: “producer” and “consumer”. The compatibility stacks include a channel to implement the communication pipe and a format to specify the kind of data that flows through the channel.
Fig. 9 shows Component A producing data as “CSV” (comma-separated values) and piping them into Component B through a “UNIX pipe” channel.

![Fig. 9: Example of pipe compatibility pattern](image)

“Component-container” compatibility pattern
This pattern models the interaction between components and containers in CBSE technologies (COM, EJB, .NET). The considered interaction primarily regards the
lifecycle management of components within a container: set-up, activation, deactivation, and dismissal. The pattern comprises two roles: “component” and “container”. Stacks are only required to specify – as a layer – the CBSE technology used in the interaction (Fig. 10).

The purpose of this pattern is to map components and containers to CBSE technologies.

3. COMPATIBILITY CHECK BETWEEN MANY COMPONENTS

Compatibility check is useful to reduce the set of candidate components chosen to build a system, working as a selection criterion, and eliminating incompatible components.

The compatibility check facility allows integrators to retrieve only compatible components from the CLARiFi repository but it requires to suppliers some additional effort. That effort is required during the insertion of the components in the CLARiFi database. The CLARiFi system helps suppliers to classify components, according to extendable and domain specific taxonomies, and to specify communication ports through stacks and layers.

Theoretically, any pair of components is candidate for interaction and can be checked for compatibility. More precisely, any pair of ports can be checked for compatibility. Since the number of possible checks grows proportionally to \( n^2 \) (where \( n \) is the number of ports), there is a complexity problem when there are many ports. Two approaches are possible.

The former approach is to automatically and extensively perform all the possible checks. Its advantage is that the broker – rather than the integrator – does most of the work. An evident disadvantage is the computational complexity. Another disadvantage is the fact that many pairs of ports logically do not need to be checked (because the functional interconnections between components do not require such pairs to exist): checking them may report compatibility where this is not needed.

The latter approach lets the integrator choose on which pairs to perform the compatibility checks. This approach focuses the checks only on the ports that are likely to interact, but a greater effort is required from the integrator. Moreover, this approach allows the integrator to focus to only on ports that he really uses in the integrated product, discarding ports that are never used therefore, it is not important if they are compatible with other ones.

The CLARiFi project adopts the second approach. This protected the implementation from the mentioned risk of a computational complexity. The CLARiFi system shows available ports to the integrator who chooses to check only the ones he really needs to build his product.

4. (PARTIAL) INCOMPATIBILITY

Two components must implement compatible stacks to communicate. In the ideal case, compatibility is achieved through identical stacks, belonging to components in complementary roles within the same compatibility pattern (Fig. 11).

There are cases where ports are not compatible or match partially. A categorization of the cases is the following:

- **Mismatch in the lower layers (the channels).** This may imply a difference in the underlying protocols or APIs. For instance, component A may be an XML-RPC server and component B may be a SOAP client. Sometimes an adapter may bridge the gap. The XML-RPC and SOAP example would probably suffice a simple adapter (or protocol bridge) due to the similarities between the two protocols. Other cases may be more complicated due to more severe differences between the protocols.

- **Mismatch of compatibility pattern.** Frequently, communication between components can be achieved conceptually using different compatibility patterns. For instance, request-response, data sharing, and pipes are equivalent for many purposes; therefore the implementation is just a design decision. An incompatibility of patterns is usually revealed also by an incompatibility in the lower layers (e.g. a request-response port would not use a
data-sharing API).

- **Mismatch in the upper layers (the data formats).** This form of incompatibility is metaphorically similar to two people who want to communicate (common semantics) over the same medium (e.g. the phone: channel layer) but speaking different languages (data format). In a real example, two components could share a table to represent sales statistics in an e-commerce application, but incompatibility could arise in the way the table is represented (e.g. Microsoft Excel vs. Comma-Separated Values). Also here an adapter or format converter could solve the incompatibility. However, the specific case and the functional objective need be considered: for instance, Microsoft Excel and CSV tables may be equivalent to represent structured data, but may be not equivalent if also the graphical formatting of data is required.

- **Mismatch in the semantics.** This form of incompatibility may result from the different understanding two components have of the same problem. A typical case is a difference in the underlying data models: one component models a relationship between two entities as one-to-many, while another component models it as many-to-many. Even compatibility in the data format layers may hide incompatibility in the semantics: e.g. two components may use the same format to represent sales, though one understands it as past sales while the other as future sales estimates. There are cases where the data formats are “semantically rich”, i.e. they express information in terms that are very close to the semantics of the problem. This is frequently the case of standards (e.g. ebXML, CBL, etc.). Other times, the data formats are “semantically poor” and the interpretation of the data is left to the business logic of the various components.

The list is proposed to exemplify possible cases of incompatibility. For integrators, the ideal form of compatibility is “proven compatibility”, where past positive experiences indicate almost certainty of integration. In reality, the integrator’s inexperience with – presumably – the majority of components in the repository determines an uncertainty on their suitability for integration. This justifies the introduction of objective technical means – such as stacks – to reduce the uncertainty on compatibility.

Compatibility proven through compatibility checks increases the probability of integration. However, the results of the checks need to be interpreted: apparently compatible stacks may hide incompatibilities that the formalism cannot express; also, apparently incompatible stacks may be easily made compatible through adaptation (“glueware”). The principle should be not to discard altogether components with (partial) incompatibility, but to evaluate the individual cases and the possible adaptations.

5. **Conclusions**

The adopted approach in this paper was to consider the technical aspects of compatibility that suppliers can provide in support of their components, that the broker can store, and that the integrator can use in the ranking and selection of components. Moreover, rather than talking of compatibility as a binary (yes/no) property, we advocate the concept of “integration effort”, as a continuous value that is desired to be as low as possible. As a technical tool, stacks are a support (also visually effective) to understand the underlying mechanisms of compatibility and to estimate, in complement to the integrator’s experience, the integration effort.