

CONFLICT PATTERNS: TOWARD IDENTIFYING SUITABLE MIDDLEWARE

L. Davis R. Gamble
Software Engineering and Architecture Team
Department of Mathematical and Computer Sciences
University of Tulsa
600 S. College Ave.
Tulsa, OK 74104

Abstract

Architectural patterns aid developers in resolving coarse-grained integration problems among components. These patterns are assembled from functionality slices that resolve various communication problems between applications. However, little attention has been paid to how interoperability problems and their resolution are embodied in these patterns. Mapping these problems to specific functionality promises insight into composing integration architectures by illuminating the consistent, high-level solutions that resolve individual conflicts. The objective of this paper is to describe patterns of interoperability conflicts along with their typical resolution in an effort to present reusable solutions for the design of integration architectures. To this end we present the Extender-Controller-Translator (ETC¹) model, and detail its use in a pattern to resolve the problem of data inconsistency.

1 INTRODUCTION

In a world where technology evolves at an exponential rate, component-based software engineering (CBSE) has become the paradigm needed for business and military development efforts to succeed. In practice, however, CBSE has not been the watershed developers hoped it would be. Interoperability conflicts often negate its advantages because independent software components, especially COTS products, may not be as open as assumed. In this paper, we focus on describing the conflicts, the architectural influences that can create them, and the basic functionality that needs to be embedded in an integration architecture to resolve them. To this end we present the Extender-Controller-Translator (ETC) model, and detail its use in a pattern to resolve the problem of data inconsistency. We use patterns for this description for two reasons. The first reason is that these interoperability conflicts are stable with respect to static architectural properties of components. This means that the same properties point to the same potential conflicts. The second reason is that the embedded functionality is common to all integration patterns. Therefore, this

manner of description provides a direct correlation to existing patterns for component integration.

2 RELATED RESEARCH

There are many issues that underlie research on component interoperability, software architecture and design patterns. Below we introduce these subjects, along with their applicability to and divergence from our work.

The *software architecture* of a system is a high-level description of its computational elements, the means by which they interact, and the structural constraints on interaction [29,32]. On the basis of *architecture mismatch*, characteristics have been specifically viewed with respect to their potential impact on interoperability [1,2,3,4,10,13,31,33]. We have expanded on this effort to classify and relate published characteristics resulting in a highly relevant set for interoperability analysis [8].

Connectors have become increasingly important in software architecture analysis, having been relegated to first-class status in many descriptions [12,16,23,25,34]. Taxonomies are being constructed (e.g., [25]) to classify and analyze a wide range of connectors and to promote the formation of unique connectors.

What makes component integration difficult are incorrect, inconsistent or impeded data and control communications expected during the execution of the composite application between the components themselves or between the expectations of the application and the intentions of the components. These problems produce three conflict categories: data transfer, control transfer and interaction initialization [8].

Common patterns that are considered integration patterns are the Proxy [5] and Broker [5,27]. A Proxy functions as a representative to a client for a component to which it wishes to communicate. The Broker affords clients and servers location transparency, plug and play mobility, and portability. More recent integration patterns, including the Wrapper-Façade, Component Configurator, and the Interceptor, represent repeatable solutions to particular integration problems [30]. These problems focus on the lack of robustness present in non-OO API's, the need for dynamism in integrated systems and the need for dynamic service addition in ORB implementations.

EAI patterns focus on the integration of heterogeneous enterprise application components [21]. Building on previous design patterns [11], these patterns describe the Integration Adapter, Messenger, Façade, and

¹ Thank you to Reviewer 1 for the suggestion of ETC.

Mediator, as well as, the Process Automator. They provide enterprise application functionality such as interface unification and reusability, location transparency and negotiation, and component decoupling. Their usefulness is limited to only specific conflicts, and require new pattern identification should other conflicts be revealed.

Mularz [27] identifies four patterns called *integration architecture patterns*: (Legacy) Wrapper, Work Flow Manager, Broker, and Shared Repository offer services similar to those provided by EAI patterns. Mularz presents a unique integration pattern language that includes keywords such as Integration Context, and Integration Conflict. However, recognizable and standardized conflicts that pinpoint the problems these patterns resolve are not present. What the ETC model provides is a pattern for a specific conflict which can be composed with other patterns to form a final integration solution. Most importantly, this solution is based on the exact interoperability conflicts present in the system, providing traceability in integration efforts.

3 FINE-GRAINED ARCHITECTURE INFORMATION

In this section, we present our prior work on architecture characteristics and connectors for use in integration.

3.1 Component-level and Application-level Characteristics

Our previous research suggests that it is feasible and desirable in interoperability analysis to partition architectural characteristics across two viewpoints: component-level and application-level [10,12,20]. *Component-level characteristics* contribute to an understanding of the exposed interface of an encapsulated component. In turn, *application-level characteristics* architecturally describe the overall configuration and

coordination requirements of the component systems when integrated into a single application.

Using extensive empirical analysis of published characteristics, we define abstraction levels and use semantic networks to identify a set of style-related characteristics [7,8,15]. We do not claim that the set is complete - more characteristics may be introduced with further research. However, our methodology allows for the direct classification and connections of new characteristics.

3.2 Foundational Elements of Integration

We model the basic functionality need for integration as three integration element connectors: *translator*, *controller*, and *extender* [16,17]. These *integration elements* supplement the traditional architecture connectors by providing specific functionality to resolve interoperability problems.

3.2.1 Translator

For a connector to be considered a *translator* (Figure 1), it must communicate with other components independent of their identities, the data (e.g., procedure calls, signals, parameters, event calls, shared data, remote procedure calls and information) on the input ports must have a uniform structure that is known by the translator's domain, the converter must be a total mathematical relation that performs the translation of the input and maps it to the output, placing it on all output ports. The relation may be a composition of relations.

3.2.2 Controller

A *controller* integration element (Figure 1) coordinates and mediates the movement of information between components using predefined decision-making

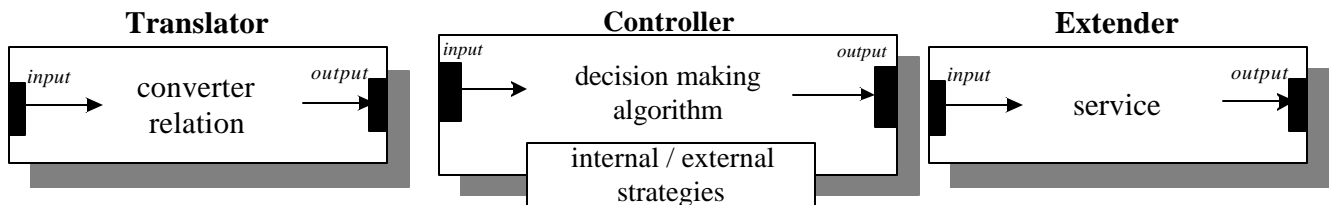


Figure 1: The Basic Translator, Controller, and Extender

processes. The basic controller can be further specified. In Section 4.2, we identify three specializations of the controller: the *Composer Controller*, the *Distributor Controller*, and the *Coordinator Controller*. The *Composer Controller* has multiple input ports, but may have any number of output ports. The composer controller makes a decision as to which input to pass, and broadcasts this to all output ports. The *Distributor Controller*, on the other hand, makes decisions on which output receives information and what information is passed. It has one or

more input ports and multiple output ports. The *Coordinator Controller* is a bi-directional controller that can both compose and distribute.

3.2.3 Extender

An *extender* integration element adds those features and functionality to an integration architecture to further adapt it to the integrated system (Figure 1). One can think of an extender as embodying those behaviors

not performed by a translator or controller such as buffering, adding callbacks, opening files, or performing security checks.

3.2.4 Integration Architecture

An integration architecture is defined to be the software architecture description, using integration elements, of a solution to interoperability problems between at least two interacting component systems [16]. An integration architecture forms the foundation of design patterns and off-the-shelf (OTS) middleware [17]. A preliminary taxonomy has been developed to illustrate the relationships between integration architectures and the design patterns and integration strategies they formulate [16]. Middleware researchers believe that requirements and conflicts must be considered when designing next generation middleware [EABCCGHKPRS98, GEI01]. Without an established link between software properties, interoperability conflicts and the resultant functionality middleware will not be flexible and adaptable.

4 CONFLICT PATTERNS

We combine and extend the pattern languages defined by Mularz [27] and Buschmann et al [5] to accommodate the information needed to express interoperability conflicts. Below we briefly define each entry in the pattern language.

4.1 Conflict Pattern Language

Name: A descriptive name for the interoperability conflict.

Category: The general category under which the conflict falls (see Section 2).

Context/Motivating Factors: A brief explanation describing the context of the problem as it pertains to the communication between components.

Problem: How the problem occurs and what architectural characteristics contribute to it. Also, a generalized example of the conflict may be given.

Solution: The integration element(s) that resolve this conflict. As part of the justification, a list of the functional requirements influencing the choice of integration elements should be included.

Structure: *Class-Responsibility-Collaborators* (CRC) diagrams of the integration architecture, or composite integration element solution.

Behavior: A diagram describing the basic behavior of the integration architecture as per the collaborators defined in the CRC cards.

Satisficing² Patterns: Existing patterns that approximate or satisfy the integration architecture. They satisfy, in

that, they contain the integration elements described in Structure. Satisficing patterns can often depict the overlap between solutions and aid developer in the choice of implementation.

4.2 The Data Inconsistency Interoperability Conflict

Name: Inconsistent Data

Category: Transfer of Data

Context/Motivating Factors: Partial, outdated, or incomplete data is communicated among components causing unexpected data inconsistencies within the application.

Problem: A component that stores its data locally, in most cases, can only share data through an explicit passing of that data. If this component must transfer its data to another component that exhibits some form of concurrency, the data may become inconsistent. This is because multiple processes may need joint access to that data.

The same conflict can occur if the application environment requires concurrent data exchange. Thus, should the components in a concurrent application attempt to communicate data without some mechanism to control simultaneous writes, there is no guarantee that what components read is the most correct information. The lost update problem, a well-identified concurrency issue in database design, is an example of the inconsistent data problem.

Solution: Utilize a *controller* to sequence concurrent access to the data in a component. Components communicate with the controller, and the controller makes a decision concerning the communication it has been given. One of the decisions a controller would make to resolve this particular conflict is what data is communicated to participating components. A second decision this controller could make is which communication can next gain access. Third, a controller can determine which component a communication can access.

The Inconsistent Data pattern allows components to make multiple requests without any alteration to their personal structure or information. Components can communicate through a coordinating representative without knowledge of what either that representative or other system components are doing.

By understanding and applying the Inconsistent Data pattern heterogeneous components can forward data and expect it to arrive at its destination correctly. This is achieved by housing it until it is ready to be sent, coordinating its destination, and translating the data into a format that its partner(s) can understand.

Structure: The Inconsistent Data pattern contains *components*, a *controller*, *translators*, and *extenders*.

A *component* is a standalone system, often a COTS product, which participates in the system to be integrated. There is no specified platform, language,

² Satisfice: to obtain an outcome that is good enough. Satisficing action can be contrasted with maximizing action, which seeks the biggest, or with optimizing action, which seeks the best. - *Herbert A. Simon, Models of Man, 1957*

interprocess communication structure, or functionality embodied by components. It simply provides functionality to the system as needed. Components whose communication provokes the Inconsistent Data conflict tend to have more dynamic structure and communication – concurrent execution is a good example.

The *controller* makes a decision concerning communicated data to prevent inconsistencies within the application. The controller can either receive or forward data to translator(s) to ensure data is in the correct format to and from a component. An instance of the Inconsistent Data pattern may need one of three types of controllers mentioned in Section 3.2.2.

A *translator* is needed which has at least one input port and one output port, though it is not restricted to one. This functionality piece provides format consistency for the communicating components. Thus, what is forwarded to components through a translator has been adapted to the correct interface.

The *extender* buffers information and reconciles it such that multiple requests are made in order, and overlapping requests are discarded. It may also provide component registration and lookup functionality to allow dynamic plug in of new components.

As can be seen under the Satisficing Patterns heading below, multiple translators/controllers/extenders may be employed in order to satisfy the exact requirements of a chosen implementation. For example, should communication be bi-directional (as with a client/server system) a second translator/controller/extender combination could be implemented for a component to reciprocate communication

The *Class-Responsibility-Collaborators* (CRC) diagrams of the *translator*, *controller* and *extender* to

resolve the Inconsistent Data conflict across multiple components are depicted below.

| | |
|--|--|
| Class Translator | Collaborators ☞ Buffer ☞ Distributor Controller |
| Responsibilities ☞ Updates incomplete data ☞ Checks data for lost updates ☞ Marshalls data | |
| Class Controller | Collaborators ☞ Component (s) ☞ Tranlator |
| Responsibilities ☞ Recieves translated data ☞ Negotiates input/output ports on which the data is put. | |
| Class Extender | Collaborators ☞ Component (s) ☞ Translator |
| Responsibilities ☞ Recieves incoming data ☞ Houses data ☞ Component registration/lookup | |

Figure 4: CRC Diagrams for ID Solution

Behavior: Below is a flow diagram of the solution for the conflict.

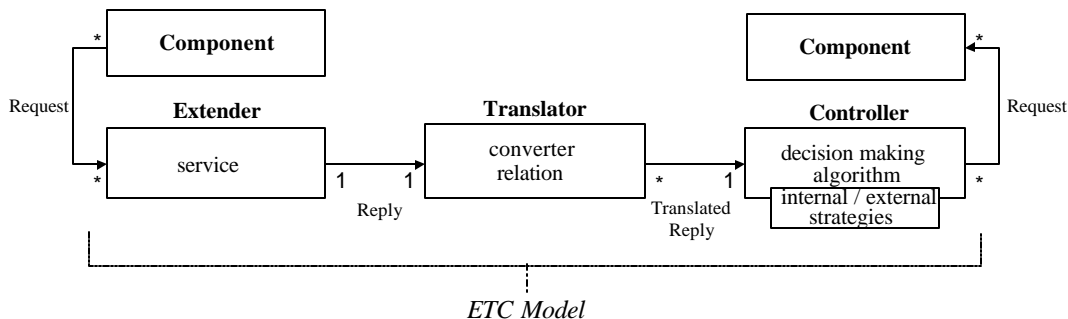


Figure 5: Inconsistent Data Solution Architecture

Satisficing Patterns: Some patterns, which nominally meet the requirements set forth by the conflict, are the *Broker* [5, 27], the *Workflow Manager* [27], and *Shared Repository* [27].

The *Broker* consists of component translator-extender pairs (a Proxy) to buffer calls and translate them into an intermediate service request that it forwards. A Coordinator Controller then finds whether the requests

can be satisfied and forwards it to the appropriate component. The Broker is a satisficing pattern such that it provides for independently executing systems while affording a means by which all components involved are getting the correct data at the correct time.

In the *Workflow Manager* the components are activated and sequenced by a Distributor Controller that captures the user input and then invokes the appropriate

sources based upon the input data. It utilizes multiple translators to convert data into each component's expected format. Finally, it employs an extender to house component profiles – information such as its address, communication protocol, its supported data exchange formats, etc. Two other extenders are present: one to house the sequencing order for each task and the other to locate participating files and applications. The workflow manager resolves this conflict because the tasks it carries out are tedious and prone to human error, thus, causing the outcome to be inconsistent or invalid.

A *Shared Repository* consists of a Composer Controller that takes in multiple requests and sequences them for access to the shared model. Associated with every component are translator pairs to convert the component's data format into and from the shared data format. The intermediate model is an extender that both uncovers and discards redundant information while buffering the data for distribution. This pattern best addresses this conflict. It does not, however, reduce the applicability of the other design patterns.

6 DISCUSSION OF APPROACH

The results of our approach can be justified in the following ways. To begin, it is evident in prior integration architecture research [27,5,21] certain software concerns – location transparency, unified interface or plug-and-play mobility - can be identified as causing similar conflicts. Furthermore, these concerns may be more closely related than once thought. In the pattern language utilized by Mularz [27], *Integration Conflict* is a defining category. She, however, does not pinpoint a definitive set of conflicts that form her result. Yet, just as the solution is repeatable, the set of conflicts from which it was defined is constant.

It seems reasonable that a conflict could map to specific functionality within the pattern. For example, should your components be written in different languages on different platforms and require that they have the exact address of the object/service they are requesting, some form of intervention would be necessary to allow them to communicate in a distributed environment and also execute independently. In the Broker pattern [5], proxies connecting the clients and servers to the broker achieve location transparency, as well as platform independence. Thus, the proxy corrects the data inconsistencies between the client and the servers it is calling. Plus, the client circumvents having to directly call said servers, avoiding transfer or location errors. Hence, conflicts define the functionality of an integration solution, and their needs can clearly translate to the desired integration architecture.

Most importantly, this approach provides two things important to both integration architecture formation and evolution. First, it provides clear reasons for the choice of a pattern. By patterning the conflicts themselves, all present interoperability problems can be

considered, and overlap between sacrificing patterns can pinpoint the best fit. Second, the patterns identify all necessary functionality to resolve a particular problem. This provides traceability in an integration and aids developers when a system evolves.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we present a static set of recurring integration conflicts to be patterned. We outline a pattern language distinctive to these conflicts that is built on established research. The pattern for data inconsistency is described with this language, and its purpose justified. Through extensive empirical study, we have identified other conflict types based on software architecture characteristic analysis. Currently, several conflicts have been defined as patterns [9]. Definition of the complete set of conflicts is on going. Further research may reveal needed modifications to the pattern language to ensure a pattern's robustness and depth. The pattern descriptions of all of the conflicts will reinforce the relevance of our findings and make less difficult the mapping to existing integration strategies.

8 ACKNOWLEDGMENTS

This research is sponsored in part by AFOSR (F49620-98-1-0217) and NSF (CCR-9988320).

9 REFERENCES

- [1] Abd-Allah, A. *Composing Heterogeneous Software Architectures*. Ph. D. Dissertation, Computer Science, University of Southern California, 1996.
- [2] Abowd, G., Allen, A., Garlan, D. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodologies*, 4(4): 319-64, 1995.
- [3] Allen, R., Garlan, D. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodologies*, 6(3): 213-49, 1997.
- [4] Barret, D., Clarke, L., Tarr, P., Wise, A. *An Event-Based Software Integration Framework* 95-048. Laboratory for Advances Software Engineering Research, Computer Science Dept., Univ. of Massachusetts, 1995 (revised 1/96).
- [5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.
- [6] Charles, J., Middleware Moves to the Forefront. *Computer*, 22(5), 1999.
- [7] Davis, L., Payton, J., Gamble, R. How System Architectures Impede Interoperability, in the proceeding of the *2nd International Workshop On Software and Performance*, 2000.

- [8] Davis, L., Gamble, R., Payton, J. The Impact of Component Architectures on Interoperability, *Journal of Systems and Software*, (to appear 2002).
- [9] Davis, L., Gamble, R., Underwood, D., Conflict Patterns: Toward Identifying Suitable Middleware University of Tulsa. Tulsa, OK, 2000.
- [10] Gacek, C. Detecting Architectural Mismatches During Systems Composition USC/CSE-97-TR-506. Center for Software Engineering, University of Southern California, Los Angeles, CA, 1997.
- [11] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [12] Garlan, D. Higher-Order Connectors, *Workshop on Compositional Software Architectures*, Monterey, CA, January 6-7, 1998.
- [13] Garlan, D., Allen, A., Ockerbloom, J. Architectural Mismatch, or Why it is hard to build systems out of existing parts. In, *17th International Conference on Software Engineering*. Seattle, WA, 1995.
- [14] Garlan, D., Monroe, R., Wile, D. ACME: An Architectural Description Language. In, *CASCON*, 1997.
- [15] Kelkar, A., Gamble, R. Understanding the Architectural Characteristics behind Middleware Choices. In, *1st International Conference in Information Reuse and Integration*, 1999.
- [16] Keshav, R., Gamble, R. Towards a Taxonomy of Architecture Integration Strategies, *3rd International Software Architecture Workshop*, 1-2, November, 1998.
- [17] Keshav, R., Architecture integration elements: Connectors that Form Middleware, M.S. Thesis, University of Tulsa, Tulsa, Oklahoma, 1999.
- [18] Larsen, G., Using patterns in the UML, *Communications of the ACM*, Vol. 42, No. 10, October 1999
- [19] Lea, D., Design patterns for avionics control systems. SUNY Oswego & NY CASE Center, DSSA Adage Project ADA GE-OSW, 1994.
- [20] Luckham, D., Vera, J. An Event-Based Architectural Definition Language. *IEEE Transactions on Software Engineering*, 21(9): 717-734, 1995.
- [21] Lutz, J. C., EAI Architecture Patterns, *EAI Journal*, March, 2000.
- [22] Magee, J., Dulay, N., Eisenbach, S., Kramer, J. Specifying Distributed Software Architectures. In, *The 5th European Software Engineering Conference*. Barcelona, Spain, 1995.
- [23] Medvidovic, N., Gamble, R., Rosenblum, D. Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms, *4TH International Software Architecture Workshop*, 2000.
- [24] Medvidovic, N., Rosenblum, D., Taylor, R. A Language and Environment for Architecture-Based Software Development and Evolution. In, *21st International Conference on Software Engineering*. Los Angeles, CA, May 16-22, 1999.
- [25] Mehta, N., Medvidovic, N., Phadke, S. Towards a Taxonomy of Software Connectors. In, *22nd International Conference on Software Engineering*, 2000.
- [26] Mowbray, T. and Ruh, W., *Inside Corba: Distributed Object Standards and Applications*, Reading, MA: Addison-Wesley, 1997.
- [27] Mularz. D., Pattern-based integration architectures. *PloP*, 1994.
- [28] Payton, J., Gamble, R., Kimsen, S., Davis, L. The Opportunity for Formal Models of Integration. In, *2nd Int'l Conference on Information Reuse and Integration*, October, 2000.
- [29] Perry, D., Wolf, A. Foundations for the Study of Software Architecture. *ACM SIGSOFT*, 17(4): 40-52, 1992.
- [30] Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York, N.Y.: Wiley & Sons, 2000.
- [31] Shaw, M., Clements, P. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In, *1st International Computer Software and Applications Conference*. Washington, D.C., 6 17, 1997.
- [32] Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- [33] Sitaraman, R. Integration of Software Systems at an Abstract Architectural Level. M.S. Thesis, Department of Mathematical and Computer Sciences: University of Tulsa, 1997.
- [34] Stiger, P. An Assessment of Architectural Styles and Integration Components. M.S. Thesis, Department of Mathematical and Computer Sciences: University of Tulsa, 1997.
- [35] Yakimovich, D., Bieman, J., Basili, V. Software Architecture Classification for Estimating The Cost Of COTS Integration. In, *21st International Conference on Software Engineering*. Los Angeles, CA, 296-302, 1999.
- [36] Yakimovich, D., Travassos, G., Basili, V. A Classification of Software Components Incompatibilities for COTS Integration, 2000.