A Structured Approach to Handling On-Line Interface Upgrades

Cliff Jones, Alexander Romanovsky, Ian Welch

Department of Computing Science
University of Newcastle upon Tyne, UK
{cliff.jones, alexander.romanovsky, i.s.welch}@newcastle.ac.uk

Abstract

The Integration of complex systems out of existing systems is an active area of research and development. There are many practical situations in which the interfaces of the component systems, for example belonging to separate organisations, are changed dynamically and without notification. In this paper we propose an approach to handling such upgrades in a structured and disciplined fashion. All interface changes are viewed as abnormal events and general fault tolerance mechanisms (exception handling, in particular) are applied to dealing with them. The paper outlines general ways of detecting such interface upgrades and recovering after them. An Internet Travel Agency is used as a case study.

1. Introduction

A "System of Systems" (SoS) is built by interfacing to systems which might be under the control of organisations totally separate from that commissioning the overall SoS. (We will refer to the existing (separate) systems as "components" although this must not confuse the question of their separate ownership.) In this situation, it is unrealistic to assume that all changes to the interfaces of such components will be notified. In fact, in many interesting cases, the organisation responsible for the components may not be aware of (all of) the systems using its component. One of the most challenging problems faced by researchers and developers constructing *dependable* systems of systems (DSoSs) is, therefore, dealing with on-line (or unanticipated) upgrades of component systems in a way which does not interrupt the availability of the overall SoS.

It is useful to contrast evolutionary (unanticipated) upgrades with the case where changes are programmed (anticipated). In the spirit of other work on dependable systems, the approach taken here is to catch as many changes as possible with exception handling mechanisms.

Dependable systems of systems are made up of loosely coupled, autonomous component systems whose owners may not be aware of the fact that their system is involved in a bigger system. The components can change without giving any warning (in some application areas, e.g. web services, this is a normal situation). The drivers for on-line software upgrading are well known: correcting bugs, improving (non-) functionality (e.g. improving performance, replacing an algorithm with a faster one), adding new features, and reacting to changes in the environment.

This paper focuses on evolutionary changes that are typical in complex web applications which are built out of existing web services; we aim to propose a generally applicable approach. As a concrete example, we consider an Internet Travel Agency (TA) [PD01] case study (see Figure 1). The goal of the case study is to build a travel service that allows a client to book whole journeys without having to use multiple web services each of which only allows the client to book some component of a trip (e.g. a hotel room, a car, a flight). To achieve this we are developing fault tolerance techniques that can be used to build such emergent services that provide a service which none of its component systems are capable of delivering individually. Of course, the multiplicity of airlines, hotel chains etc. provides redundancy which makes it possible for a well-designed error-recovery mechanism to survive temporary or permanent interruptions of connection but the interest here is on surviving unanticipated interface changes. As not all the systems in our system of systems are owned by the same organisation, it is inevitable that they will change during the lifetime of the system and there is no guarantee that existing clients of those systems will be notified of the change.

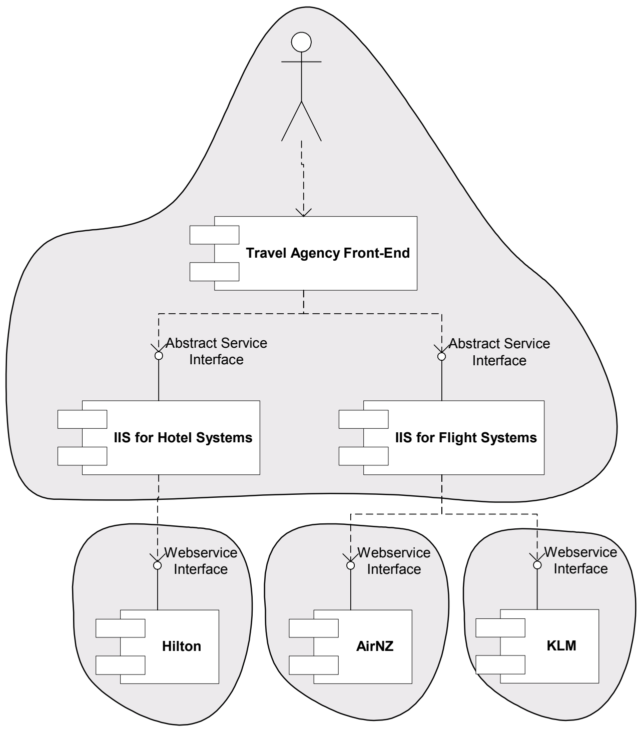

Figure 1 UML Component diagram showing the component systems that make up the Internet Travel Agency (TA). The grey areas indicate the fact that the component systems are under the control of different organisations. A user is shown interacting with the Travel Agency Front-End (TAFE). The TAFE is composed of multiple Intermediate Interfacing Subsystems (IIS). Each IIS provides an abstract service interface for a particular service type, for example the Flight Systems IIS provides an abstract service interface for booking flights with systems such as AirNZ and KLM even though each of these systems has different webservice interfaces.

When a component is upgraded without correct reconfiguration or upgrading of the enclosing system, problems similar to ones caused by faults occur, for example: loss of money, TA service failures, deterioration of the quality of TA service, misuse of component systems. Changes to components can occur at both the structural and semantic level. For example changes of a component system can result in a revision of the units in which parameters are measured (e.g. from Francs to Euro), in the number of parameters expected by an operation (e.g. when an airline introduces a new type of service), in the sequence of information to be exchanged between the TA and a component system (e.g. after upgrading a hotel booking server requires that a credit card number is introduced before the booking starts). In the extreme, components might cease to exist and new components must be accommodated.

Although some on-line upgrading schemes assume that interfaces of components always stay unchanged (e.g. [TT02]), we believe that in many application areas it is very likely that component interfaces will change and that this will happen without information being sent to all the users/clients. This is the nature of the Internet as well as the nature of many complex systems of systems in which components have different owners and belong to different organizations as shown in Figure 1. In some cases of course, there might be an internal notification of system changes but the semantics of the notification system might not be externally understood.

Although there are several existing partial approaches to these problems, they are not generally applicable in our context. For example, some solutions deal only with programmed change where all possible ways of upgrading are hard-wired into the design and information about upgrading is always passed between components. This does not work in our context in which we deal with pre-existing component systems but still want to be able to deal with interface upgrading in a safe and reasonable fashion. Other approaches that attempt to deal with unanticipated or evolutionary change in a way that makes dynamic reconfiguration transparent to the TA integrators[1] may be found in the AI field. However, our intention is not to hide changes from the application level. Our aim is to provide a solution that is application-specific and reliant on general approaches to dealing with abnormal situations. In particular, we will be building on existing research in fault tolerance and exception handling which offer disciplined and structured ways of dealing with errors of any types [C95] at the application level.

Our overall aim is to propose structured multi-level mechanisms that assist developers in protecting the integrated DSoSs from interface changes and, if possible, in letting these DSoSs continue providing the required services.

2. System Model

Integrators compose a DSoS from existing components (systems) that are connected by interfaces, glue code and additional (newly-developed) components where necessary. An interface is a set of named operations that can be invoked by clients [S97]. We assume that the integrators know the component interfaces. Knowledge of the interfaces can be derived from several sources:

[1] We use terms TA integrators and TA developers interchangeably.

interfaces can be either published or discovered (there are many new techniques emerging in this area), programmer's guides, interfaces are first-class entities in a number of environments such as interpreters, component technologies (CORBA, EJB), languages (Java).

Besides integrators there are other roles played by humans involved in the composed system at runtime, for example: clients of the composed system, other clients of the components, etc.

We assume that component upgrade is out of our control: components are upgraded somehow (e.g. off-line) and if necessary their states are consistently transferred from the old version to the new version.

3. The Framework

3.1. Structured Fault Tolerance

We propose to use fault tolerance as the paradigm for dealing with interface changes: specific changes are clearly abnormal situations (even if the developers accept their occurrence is inevitable), and we view them as errors of the integrated DSoS in the terminology accepted in the dependability community [L95]. In the following we focus on error detection and error recovery as two main phases in tolerating faults.

Error detection aims at earlier detection of interface changes to assist in protecting the whole system from the failures which they can cause. For example, it is possible that, because of an undetected change in the interface, an input parameter is misinterpreted (a year is interpreted as a number of days the client is intending to stay in a hotel) causing serious harm. Error recovery follows error detection and can consist of a number of levels: in the best case dynamically reconfiguring the component/system and in the worst with a safe failure notification and off-line recovery.

Our structured approach to dealing with interface changes relies on multilevel exception handling which should be incorporated into a DSoS. It is our intention to "promote" multilevel structuring of complex applications to make it easier for developers to deal with a number of problems, but our main focus here is structured handling of interface changes. The general idea is straightforward [C95]: during DSoS design or integration, the developer identifies errors that can be detected at each level and develops handlers for them; if handling is not possible at this level, an exception is propagated to the higher level and responsibility for recovery is passed to this level. In addition to this general scheme, study of some examples suggests classifications of changes which can be used as check lists.

3.2. Error Detection

In nearly all cases, there is a need for meta-information to detect interface changes. Such meta-information is a non-functional description of the interfaces (and possibly of their upgrades), which may capture both structural and semantic information. Some languages and most middleware maintain structural meta-information, for example Java allows structural introspection and CORBA supports interface discovery via specialised repositories. However, at present there is little work on handling changes to semantic meta-information.

Meta-information for a component includes descriptions of:

- call points (interfaces), including input parameters (types, allowable defaults), output parameters (types, allowable defaults), pre- and post-conditions, exceptions to be propagated
- protocols: the sequences of calls to be executed to perform specific activities (e.g. cancel a flight, rent a car). A high-level scripting language can be used for this.

Interface changes can be detected either by comparing meta-description of old and new interfaces or if a component supports some mechanism to notify clients of changes. Another, less general, and as such less reliable, way of detecting such changes is by using general error detection features (some reasonable run-time type checking; pre- and post-conditions, or assertions of other types of checking parameters in the call points; protective component wrappers, etc.).

The intention should be to associate a rich set of exceptions with structural and semantic interface changes (changing the type of a parameter, new parameters, additional call points, changing call points, changing protocols, etc.); this would allow the system developers to handle them effectively.

3.3. Error Recovery

Error recovery can be supported through the use of:

- different handlers (at the same level) for different exceptions related to different types of interface changes
- multilevel handling.

3.3.1. Different Handlers. System developers should try and handle the following types of exception:

- changes of types of parameters, new parameter, missing parameter, new call point
- changes of the protocols, re-ordering, splitting, joining, adding, renaming and the removal of protocol events

- change of the meta-description language itself (if components provide us with such a meta-description of its interface)
- raising of new exceptions, if the protocol changes then new exceptions may also be raised during its execution.

To provide some motivational examples, consider the Travel Agent case study.

- A very simple interface change is where the currency in which prices are quoted changes. In this case, simple type information could show, for example, that the TA system requires a price in Pounds Sterling and the Car rental is being quoted in Norwegian Crowns. An exception handler can ask for a price in Euros which might be countered with an offer to quote in Dollars. Note that this process is *not* the same as reducing everything to a common unit (dollars?), finding agreement earlier can result in real savings in conversions.
- A previously functioning communication from the TA system to a hotel reservation system might raise an exception if a previously un-experienced query comes back as to whether the client wants a non-smoking room. Either of two general strategies might help here: the query could come marked with a default which will be applied if no response is given (an exception handler could accept this option) or the coded value might (on request from the exception handler) translate into an ASCII string which can be passed to the client for interpretation.
- Some of the most interesting changes and incompatibilities are likely to be protocol changes. An airline system might suddenly start putting its special offers before any information dialogue can be performed; the order in which information is exchanged between the TA and its suppliers of cars, flights etc. might change. Given enough meta-information, it is in principle, possible to resolve such changes but this is far more complex than laying out the order of fields in a record: it is the actual order of query and response which can evolve.
- In the extreme, the chosen meta-language might change. Even here, a higher-level exception handler might be able to recover if the meta-language is from a know repertoire.
- When an airline ceases to respond (exist?) the TA system must cope with the exception by offering a reduced service from the remaining airlines.
- Communication with new systems might be established if there is some agreement on meta-languages which can be handled.

In all of the above cases, the attempt is to use exception handling to keep the TA system running. Of course, notification of such changes might well be sent to developers; but the continuing function of the TA should not await their availability.

3.3.2. Multilevel Handling. Exceptions are propagated to a higher level if an exception is not explicitly handled or an attempt to handle the exception fails. This leads to a recursive system structuring with handlers being associated with different levels of a system. Possible handling strategies are:

- request a description of the new interface from the upgraded component
- renegotiate the new protocol with the component
- use a default value of the new parameters
- pass the unrecognised parameters to the end client (e.g. in ASCII)
- involve system operators into handling
- exclude the component from the operation
- execute safe stop of the whole system.

When designing handlers DSoS developers can apply the concepts of backward recovery, forward recovery or error compensation [L95]. Backward recovery restores the system to its state before the error, for example the TA abandons (aborts) a set of partial bookings making up an itinerary if one of the components cannot satisfy a particular booking. Forward recovery finds a new system state from which the system can still operate correctly, for example where DSoS developers decide to involve people in handling interface changes: TA support/developers, TA users/clients, component support. Error compensation relies upon the system state containing enough redundancy to allow the masking of the error. An example of error compensation is the use of redundant components. For example, in the TA case study if the KLM server changes its interface and TA cannot deal with this, it ignores it but continues using servers of BA and AirFrance.

After the TA has been safely stopped or a component has been excluded, the TA support and developers can perform off-line analysis of the new interface of the component (cf. fault diagnosis in [L95]).

4. Related Work

The distributed computing community has considered the problems of maintaining meta-information for service discovery within the context of loosely coupled distributed systems such as DSoSs. Most middleware systems implement some form of object trading service, for example CORBA has an Object Trader Service, Jini has a Lookup Service, and .NET uses services provided by the Universal Discovery, Description and Integration (UDDI) project. Object traders enable providers to advertise services by registering offered interfaces with a trading service. Clients locate a service by querying the trader using descriptions based on the structure of an interface and quantitative constraints [S97]. As with our proposed solution, object traders provide the ability to associate some meta-information with services. However, there is an assumption that once a client has found a service that uses a particular interface then that interface will remain static. Another difference is that we plan to maintain a richer set of meta-information with services that capture both structural and semantic information about interfaces such as versioning information, protocols etc.

On the other hand, the object oriented database community have explicitly considered system evolution. They have developed schemes for schema evolution, schema versioning and class versioning. For example, in [AF00] schemata of multiple DBs are expressed in XML. In this approach the user's queries are written using a domain standard, that identifies the various entities and relationships, and for each data-source/base there is a mapping from that source entities to the domain standard. So, that a rewriting of the user's query to the various source formats can be done automatically. Our work differs in that in addition to structural changes we consider semantic changes such as protocol mismatches that occur when evolution takes place. Also the solutions proposed by this community tend to assume the existence of a centralised authority for enforcing control whereas we are working in the context of decentralised authority.

There has been some work on resolving protocol mismatches in the area of component based development. In [W02] the concept of a component adaptor is introduced. It describes adaptations of the external behaviour independently of a specific API. When the adapter is applied to a composition of components the required adaptations can be automatically inserted. This is achieved through the application of algorithms that are based on finite automata theory. Our work differs in that we consider dynamic rather than build-time changes to protocols and we consider more wide ranging adaptation than just the renaming or addition of protocol events.

When implementing our proposed solution we intend to exploit this related work and some other features provided by modern component-oriented technologies and Internet technologies. Other useful features that can be used are language support for runtime reflection [IW02], interface repositories and type libraries, and services such as CORBA's Meta-Object Facility that defines standard interfaces for defining and manipulating meta-models.

5. Conclusions

This paper has not proposed a totally general or efficient solution; our interest is in providing a pragmatic approach that explicitly uses a fault tolerance framework. Our work is motivated by real problems encountered when considering a case study where mismatches due to evolution must be dealt with at runtime. Although there are some existing approaches to this problem we do not try to hide evolution from the application developer but provide a framework for dealing with it dynamically.

Acknowledgements. Our thanks go to Panos Periorellis and Christos Kloukinas for their helpful comments. This work is partially supported by European IST DSoS Project (IST-1999-11585) and by Dependability Interdisciplinary Collaboration funded by EPSRC/UK. Ian Welch is supported by European IST MAFTIA Project (IST-1999-11583).

References

[AF00] B. Amann, I. Fundulaki, and M.Scholl. Integrating ontologies and thesauri for RDF schema creation and metadata querying. International Journal of Digital Libraries, 3, 3, pp. 221–236, 2000.

[C95] F. Cristian. Exception Handling and Tolerance of Software Faults. In Lyu, M.R. (ed.): Software Fault Tolerance. Wiley, 1995, pp. 81-107.

[IW02] I. Welch. A Reflective Security Architecture for Applications. PhD Thesis. Department of Computing, University of Newcastle upon Tyne (in preparation).

[L95] J.-C. Laprie. "Dependable Computing: Concepts, Limits, Challenges". Proc. of the 25th Int. Symposium On Fault-Tolerant Computing. IEEE CS Press. Pasadena, CA. 1995. pp. 42-54

[PD01] P. Periorellis, J.E. Dobson. Case Study Problem Analysis. The Travel Agency Problem. Technical Deliverable. Dependable Systems of Systems Project (IST-1999-11585). University of Newcastle upon Tyne. UK. 2001. 37 p. www.newcastle.research.ec.org/dsos/

[S97] C. Szyperski. Component Software. ACM Press. 1997.

[V02] W. Vanderperren. A Pattern Based Approach to Separate Tangled Concerns in Component Based Development. Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastucture Software, held in conjunction with the First International Conference on Aspect-Oriented Software Development (AOSD 2002). pp. 71-75. 2002.

[TT02] A.T. Tai, K.S. Tso, L. Alkalai, S.N. Chau, W.H. Sanders. Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond. IEEE TC-51, 2, pp. 121-137. 2002.