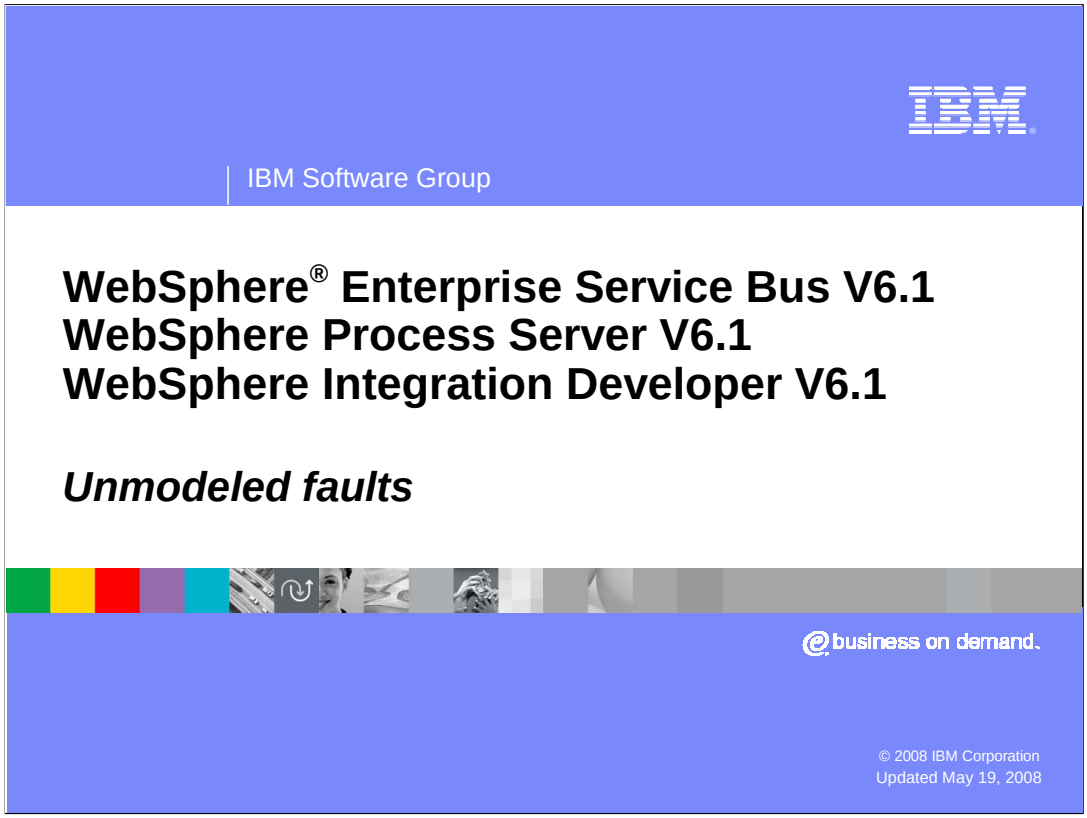
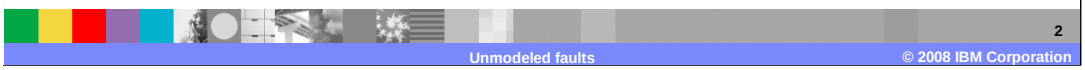


This presentation provides a detailed look at unmodeled faults and how they are handled in mediation flows.

Goals

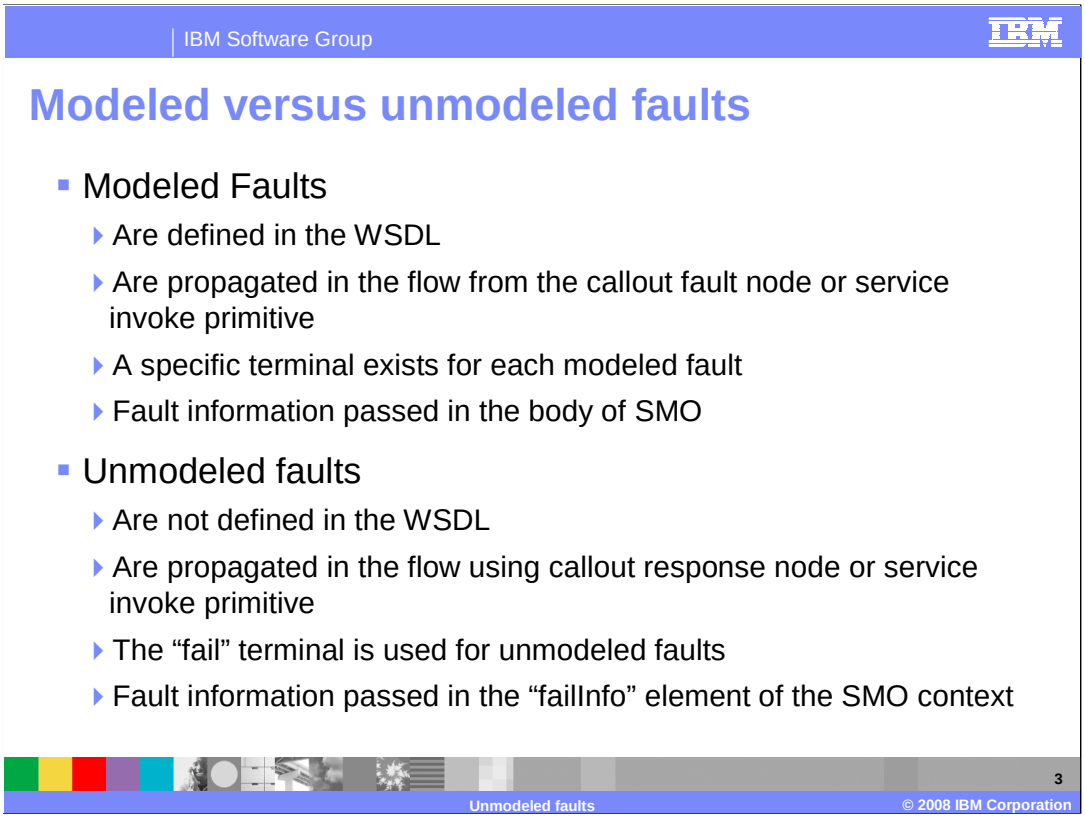
- Introduce unmodeled faults
  - Overview of function and behavior
  - Configuration of the mediation flow
  - Content of service message object (SMO)
    - Contrast modeled and unmodeled faults
  - Examples



The goal of this presentation is to provide you with a full understanding of unmodeled faults. The presentation begins with an overview of the function and behavior of unmodeled faults and then looks at how to configure your mediation flow in WebSphere Integration Developer.

You will see how both the unmodeled fault and modeled fault information are represented in the service message object, or SMO.

Finally, some examples are provided. These show you how you can make use of the unmodeled fault capabilities within your mediation flows.



One of the best ways to understand unmodeled faults is to compare and contrast them with modeled faults.

A modeled fault is explicitly defined for the operation in the WSDL for the interface. When a service returns a modeled fault, the fault is propagated in the mediation flow from either the callout fault node, or from a service invoke primitive. Which one depends upon whether the service was called from the callout node or from a service invoke primitive. For each fault defined on the operation there is a terminal with a message type specific to that fault. The fault data is carried in the body of the SMO.

In contrast, an unmodeled fault is one that is not defined in the WSDL. It is propagated back to the flow through either the callout response node or the service invoke primitive, again according to how the service was originally invoked. Because there is no fault defined in the WSDL, there can not be a unique terminal, and therefore the fault is propagated through the fail terminal. The fault data is populated in the failInfo element, which is in the context of the SMO.

Basics of unmodeled faults

- Unmodeled faults returned in the flow only when:
  - The “fail” terminal is wired
  - Retry count for unmodeled faults has been exhausted
- Handling of unmodeled faults is at your discretion
  - Wire a flow from “fail” terminal with logic to handle
  - Leaving unwired results in a mediation exception
- Message type of the “fail” terminal defined by
  - Callout response node: “in” terminal of the callout node
  - Service invoke: “in” terminal of the service invoke

This slide provides some characteristics of unmodeled faults.

First, unmodeled faults do not get propagated back to the flow except in certain circumstances. The first thing is that the fail terminal must be wired. The other consideration is the retry count for unmodeled faults that is configured for a service invoke or callout node. The unmodeled fault will only be propagated back to the flow once the retry count has been exhausted.

The next thing to point out is that the handling of unmodeled faults in the flow is entirely at your discretion. If you want to handle the fault, there must be flow logic wired from the fail terminal that performs your application specific requirements for fault handling. The alternative is to leave the fail terminal unwired, in which case a mediation exception is thrown and the flow is terminated.

The message type of the fail terminal is the same type as the input message. Specifically, the fail terminal of the callout response node will have the same message type as the in terminal of the callout node. Likewise, the fail terminal of a service invoke is the same message type as the service invoke’s in terminal

SMO contents for unmodeled faults

SMO element	Callout response node	Service invoke
“failInfo” element populated in context	yes	yes
contains body of request message	controlled by property setting	yes
contents of transient context maintained	no	yes
contents of shared context maintained	no	yes
contents of correlation context maintained	yes	yes

This table provides information about the contents of the SMO when an unmodeled fault is returned to the flow. The handling is very similar for a callout and for a service invoke, but there are some differences.

In both cases, it is the failInfo element in the SMO context that is populated with information about the fault.

When being returned from a service invoke, the body of the original message is contained in the fault message. However, when propagated through a callout response node, the original body content might not be present. This is controlled by a property setting on the call response node. If you have logic in your response flow that makes use of the original message body you must set this property. If not, there is a performance benefit by not setting the property because it eliminates the need to save the original message across the callout processing.

The remaining three are the transient, shared and correlation contexts that you configure for your flow. In the case of the service invoke, all three contain the same content as the original request message. However, in the case of the callout response node, only the correlation context contains the content from the original request. This behavior is consistent with the defined behavior for these three contexts.

Approaches to handling unmodeled faults

- Unmodeled fault handling depends on your application requirements
- Log a message or raise an event, then stop the flow
- Map to a defined fault to return to the caller
  - Interface needs a fault defined, for example “UndefinedFault” or “OtherFault”
  - Use a primitive that changes the message type of the SMO to the fault message type
  - Wire the result to an input fault node to be returned to the caller
- Recover from the fault and continue the flow
  - Use a primitive that changes the message type of the SMO to the message type of a successful result
  - Possibly log a message or raise an event
  - Wire the result back into the normal flow

There are several ways that you can handle unmodeled faults in your flow. The determination of which to use is based on your application requirements. This slide suggests three possible approaches you might want to consider.

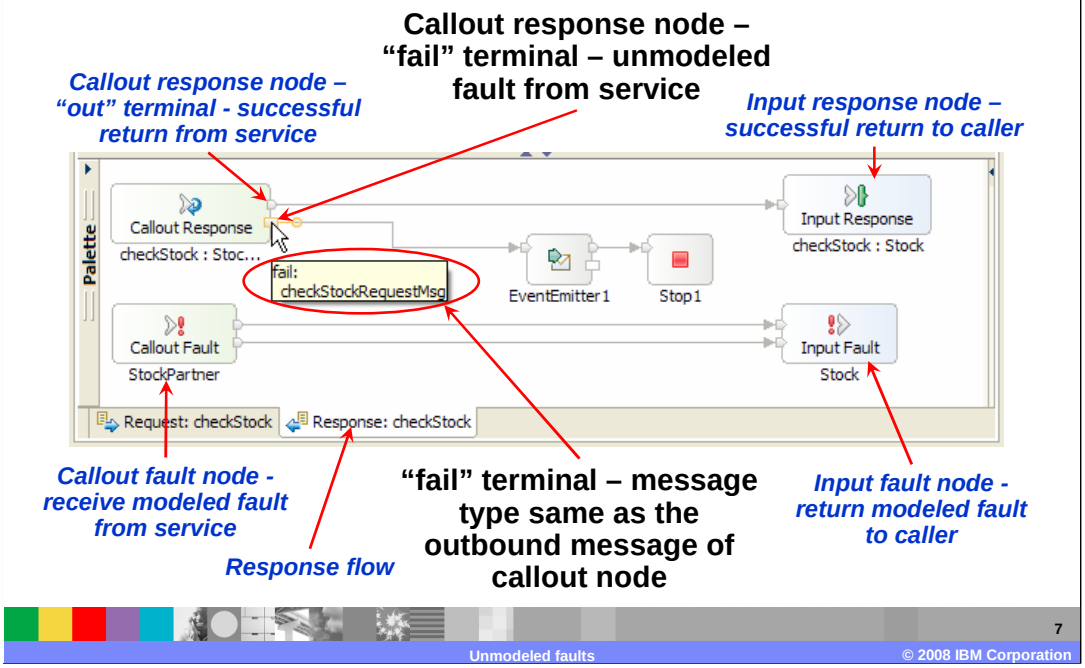
The first suggested approach is to log a message using a message logger primitive or raise an event using the event emitter primitive. Which you do is based on the approach you take towards error reporting. Following this, the flow can then be terminated using a stop primitive.

Another approach is to convert the unmodeled fault into a modeled fault that can be returned to the caller of the mediation. To do this, some modeled fault needs to be defined for the operation in the WSDL. For example, a fault named something like UndefinedFault or OtherFault. Then, in the flow, you use a primitive such as XSL transformation or business object map to transform the message type of the SMO. It can populate the message body of the defined fault with appropriate information about the unmodeled fault. This can then be wired to the input fault node that will return the fault to the requestor.

In some cases, the unmodeled fault can occur on a service call that is not critical to the overall completion of the flow. When this is the case, a flow logic designed to recover from the fault can be implemented. Typically this requires an XSL transformation or business object map primitive that transforms the message type to the message type that is normally returned when the call is successful. If needed, this primitive can set values into fields in the message that simulate a successful call. At this point, you might want to write a log message or raise an event to indicate that this error recovery processing has taken place. Then wire the recovery flow into the normal flow path at an appropriate point to continue the flow processing.

There are other possible approaches that can be taken. You need to decide what approach is appropriate for your application requirements.

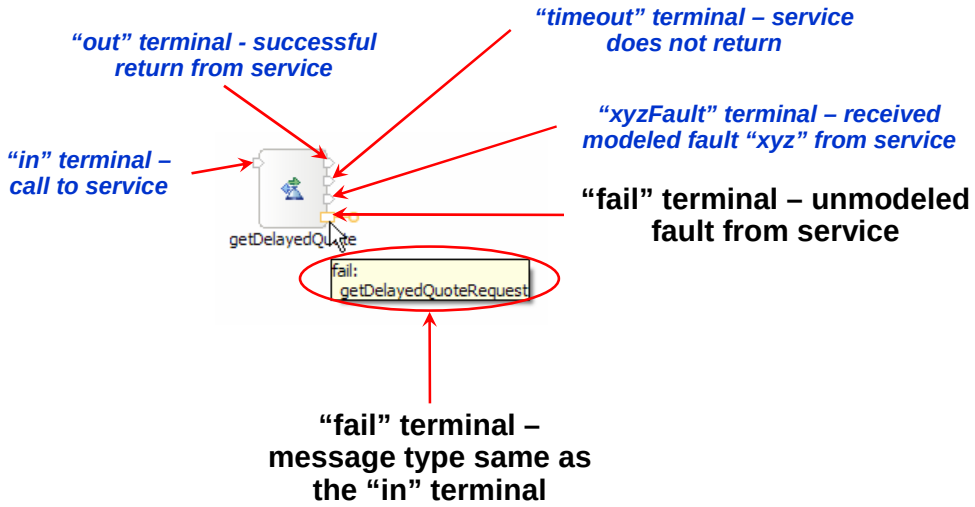
Tool support – Callout response node



This slide shows a response flow that is handling an unmodeled fault by raising an event and then stopping the flow without raising an exception.

On the top left you can see the callout response node. The terminal on the top of the node is the out terminal for a normal return, and you can see that it is wired directly to the input response node, which returns to the original caller. On the bottom left there is a callout fault node. This node is where WSDL defined, or modeled faults, enter the response flow. You can see that the terminals are wired directly to the input fault node, which returns the fault to the original requestor. Returning back to the top left, the bottom terminal of the callout response node is the fail terminal where unmodeled faults are returned. You can see that the message type for the fail terminal is checkStockRequestMsg, the same as the message type of the outbound message of the callout. The fail terminal is wired to an event emitter primitive, which produces an event to the common event infrastructure, and then proceeds to the stop primitive, which terminates the flow without raising an exception.

Tool support – Service invoke primitive

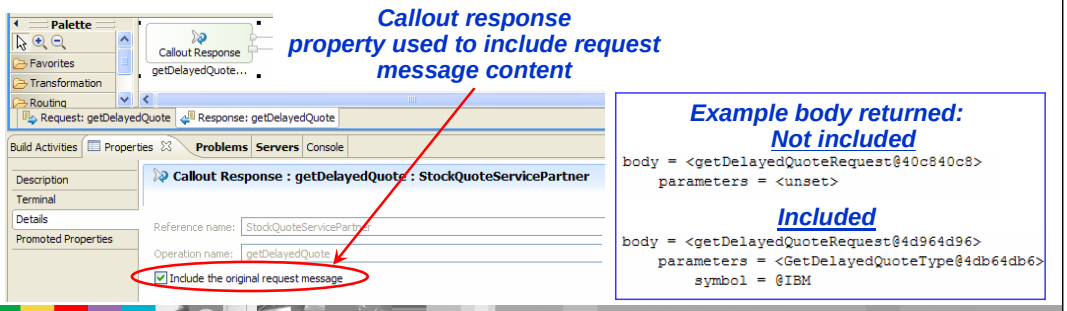


This slide shows a service invoke primitive by itself, rather than in the context of a flow. It identifies all of the terminals, highlighting the fail terminal used for unmodeled faults.

On the left of the service invoke primitive is the in terminal that receives the message used to call the service. On the right are several terminals. The out terminal is used for a successful response from the service. The timeout terminal is used when an asynchronous service request does not complete within the designated time limit. For every fault defined on the operation being called, there is a corresponding terminal, such as the xyzFault example shown in the screen capture. Finally, the fail terminal is where an unmodeled fault is returned. The message type of the fail terminal, shown here as getDelayedQuoteRequest, is the same message type as that of the in terminal.

Tool support – Including original message

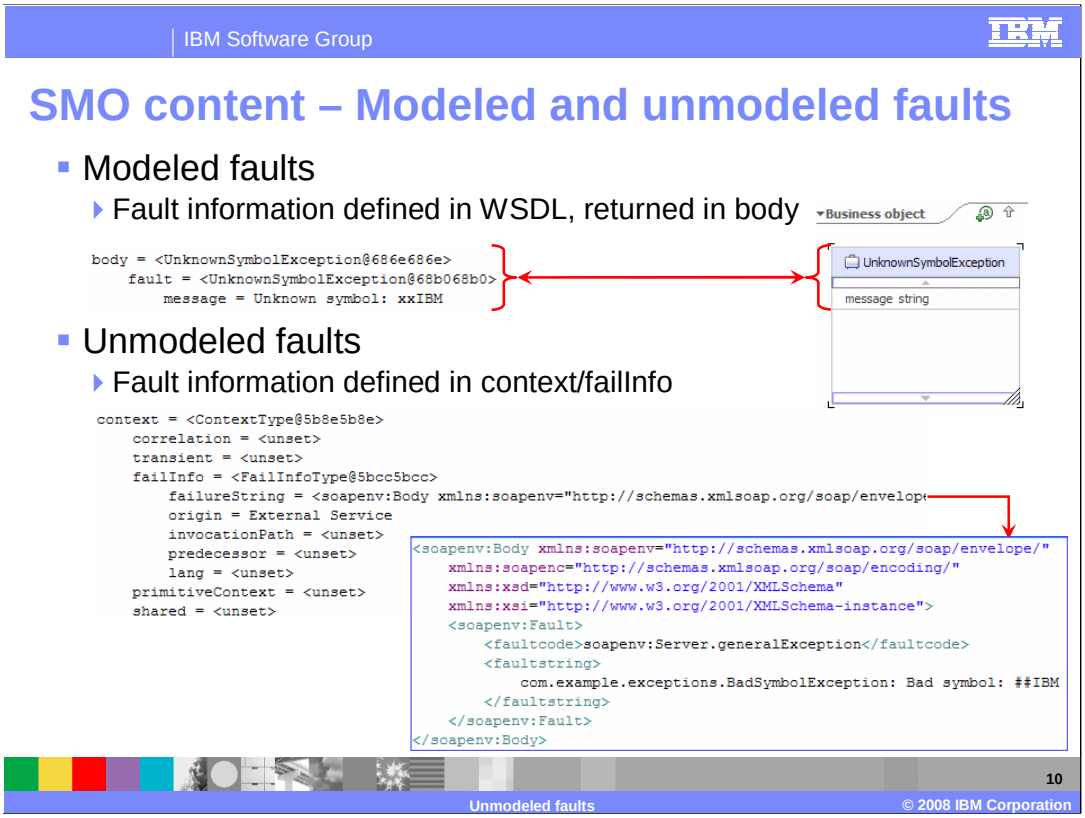
- Original request message included for service invoke
- For callout response node, controlled by a property setting
  - By default, this property is not selected
    - Performance advantage
    - Requires the entire outbound message to be saved across the call
  - Should only be included if used in the unmodeled fault flow



When an unmodeled fault is returned, it varies whether the complete original message body is included. It will always be present for a service invoke, but is controlled by a property setting for the callout response node, as is illustrated here.

The screen capture shows the response flow on the top left, and immediately underneath it is the Details panel of the Properties view for the callout response node. It is the check box labeled “Include the original request message” that controls this capability. By default, this property is not selected. There is a performance implication when the entire outbound message needs to be saved across the call. Therefore, you should only make use of this capability if your flow logic for the unmodeled faults makes specific use of the outbound message data.

On the right are screen captures showing the difference between the message bodies when the original message is not included and when it is included.

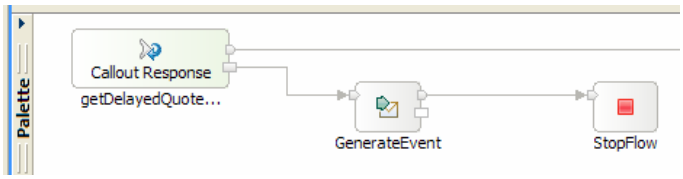


Both unmodeled and modeled fault data are placed into the SMO, but where they are placed is different. For a modeled fault, there is a WSDL definition that defines what is returned with the fault. In the upper screen capture, on the right, is a business object that has been defined to be returned for a modeled fault. On the upper left you can see that the information is placed into the body of the SMO.

For unmodeled faults, there is no WSDL definition and therefore no definition of what to place into the message body. The lower screen capture shows that the unmodeled fault is returned in the failInfo element within the context of the SMO. You can see that in this case, since the callout was for a Web service, the unmodeled fault information is returned as a SOAP fault that is placed into the failureString of the failInfo element.

Usage scenario – Generate event

- Unmodeled fault flow generates event
  - Generated with event emitter primitive
    - Handled by common event infrastructure (CEI)



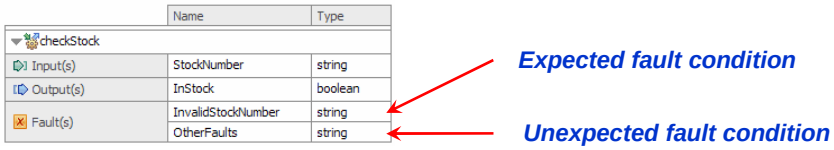
  - Event can be used to document occurrence of the fault
  - Monitoring application can take action on event
    - Use CEI to filter events
    - Obtain this event as a JMS message
    - Read this event from the event database
    - Initiate action to debug or retry operation based on event

In this example, an unmodeled fault causes an event to be generated, and then the flow terminates without an exception being thrown. You can see that the fail terminal is wired to an event emitter primitive and then it is wired to a stop primitive.

Because event emitter primitives generate common base events handled by the common event infrastructure, or CEI, you have several options regarding how you want to handle the event. The CEI server provides filtering capabilities that are used to determine what it does with the event. For example, the event can be added to the CEI event database and it can also be sent as a message to a JMS queue or published to a JMS topic. This enables a monitoring application that is searching the database or receiving the JMS message to act on the event appropriately, such as initiating debugging activities or retrying the operation.

Usage scenario – Convert to modeled fault

- Return unmodeled faults to caller as modeled fault
  - Operation defines faults for
    - Expected fault conditions
    - One additional fault for unexpected conditions

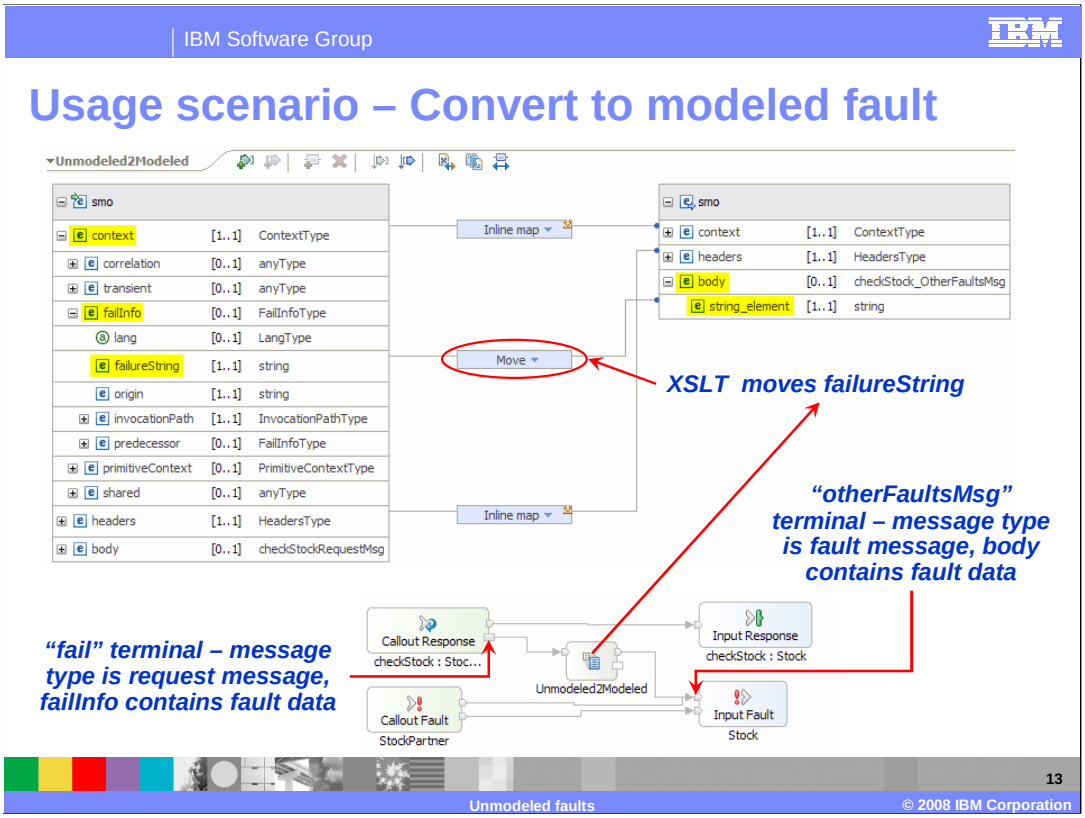


  - Flow transforms unmodeled fault to modeled fault
    - Use XSL transformation or business object map primitive
    - Transform the input request message to response fault message
    - Copy the error information from the context/failInfo/failureString to the fault body
    - Screen capture on next slide →

This next example is presented over two slides. This is the case of converting an unmodeled fault to a modeled fault so that it can be returned to the original caller.

Looking at the screen capture, you can see an interface with an operation that has an input, an output and two faults. There is an InvalidStockNumber fault that represents an expected fault condition that the service returns. In addition, there is another fault, named OtherFaults, that is used to return any unmodeled faults that occur. An XSL transformation primitive is used to transform the message from the request flow message type to the OtherFaults message type. The transformation includes copying the failureString from the failInfo section of the message context to the body of the fault message.

The next slide provides an illustration of how this is handled.



On the bottom portion of the slide is the response flow. On the bottom of the flow you can see that the callout fault node has two terminals, one for each of the defined faults. These are wired to the two terminals on the input fault node that are for the same two WSDL defined faults. These wires represent the flow that occurs if the service explicitly returns either of these two faults. In addition, the fail terminal of the callout response node is wired to an XSL transformation primitive, which is then wired to the OtherFaults terminal of the input fault node. The unmodeled faults flow through here. The upper portion of the slide shows, in the XML mapping editor, the map that is used to copy the unmodeled fault information from the failureString in the context to the body of the fault message.

Summary

- Examined unmodeled faults
  - Overview of function and behavior
  - Configuration of the mediation flow
  - Content of the SMO for modeled and unmodeled faults
  - Reviewed examples

In this presentation you were provided with an introduction to unmodeled faults. The presentation began with an overview of the function and behavior of unmodeled faults and then looked at how to configure your mediation flow in WebSphere Integration Developer. A description of how fault information is represented in the SMO for both modeled and unmodeled faults was provided. Finally, there were a couple of examples to show how you can make use of the unmodeled fault capabilities within your mediation flows.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WBIV61_UnmodeledFaults.ppt

This module is also available in PDF format at: ../WBIV61_UnmodeledFaults.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM WebSphere

A current list of other IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.