Using flows in information integration

by F. Leymann
D. Roller

Information integration has two fundamental aspects, data integration and function integration. Function integration is based on flow technology and adapter technology, and both of these add powerful capabilities to information integration. They provide access to a huge variety of data sources, such as standard applications, home-grown backend systems, and Web services. For accesses that are not restricted to read operations, flows can help in managing units of work across these data stores. When a database system is coupled with a flow engine, all of these capabilities are made available to database applications.

Information integration has two fundamental aspects: data integration and function integration. In a nutshell, *data integration* deals with the problem of making heterogeneous, "external" data sources accessible via a common interface and an integrated schema: users should perceive the collection of data as being managed by a single database system. Considerable research has been undertaken in this area, under the umbrella of federated databases,[1] and products as well as standards are being developed.[2] *Function integration* deals with the problem of making local functions from disparate applications available in a uniform manner:[3] users should perceive a homogeneous collection of functions as a base for manipulating data encapsulated by the various applications. Notably, very little research has been undertaken in this area, but a number of vendors offer products and standards[4] are appearing for enterprise application integration (EAI).

In this paper we discuss the use of flow technology as a fundamental ingredient for function integration as well as function aggregation, and as a means to bring data integration technology and function integration technology together. In the next section we review the evolution of adapter technology and its use of flow technology. The programming model resulting from the use of flows is described in the following section. An advanced use of flows in function integration scenarios is described in the next section, followed by a section on how flow technology can be used in data integration scenarios. In the final section, we describe transaction management capabilities that are made available by using flows in database environments.

Adapters

Today, applications are not perceived as independent, isolated conglomerates of functions and data but as parts that are subject to integration into a whole in support of a company's business processes. Functions performed in one application imply functions that must be performed in other applications to maintain overall consistency. For example (see Figure 1), when an order is entered into an SAP application, the corresponding available credit may

have to be modified in an Oracle application, information about the customer updated in a Siebel system, and a CICS* (Customer Information Control System) transaction run to change an entry in a database.

In such scenarios, the notion of "application" covers the spectrum from home-grown legacy applications, over standard applications (enterprise resource planning [ERP], customer relationship management [CRM], etc.), to database systems, and may even include functions provided by external business partners. Often these applications are collectively referred to as an *enterprise information system* (EIS). The meaning of "integration" ranges from uniform access to data managed by any of these applications, over uniform access to discrete functions of the applications, to the ability for changes in one application to cause automatic changes in other applications in accordance with business rules. The corresponding technology area is referred to as *enterprise application integration* (EAI) technology.

The fundamental element of any EAI solution is an *adapter* (often called *connector*—although a connector is defined with a particular semantics[4]). An adapter accepts data in the particular format of one application (*source format*), transforms the data into the format of another application (*target format*), and knows how to pass the transformed data to the second application for further processing. For example, an adapter may accept a purchase order as defined by RosettaNet,[5] transform it into a corresponding SAP intermediate document (IDoc),[6] and then use the SAP remote function call (RFC)[7] mechanism, with the created IDoc as parameter, to call the appropriate SAP function to process the purchase order.

This kind of adapter is often called a *target adapter*, because its purpose is to invoke functions of a target application to update the application's database or to query some data from it. A second kind of adapter is called a *source adapter*—it typically hooks into its associated application to capture events that are of interest to other applications. The technique used by source adapters to hook into an application depends on the application itself and varies from database triggers to publish/subscribe mechanisms. Whenever an event occurs that might be of interest to another application, the source adapter transforms the corresponding data into an appropriate format and sends it to the interested applications. Typically, a source adapter sends the data through a separate piece of middleware (see Figure 2). There are many different names for this middleware: we call it a *hub*.

Figure 1 The EAI problem

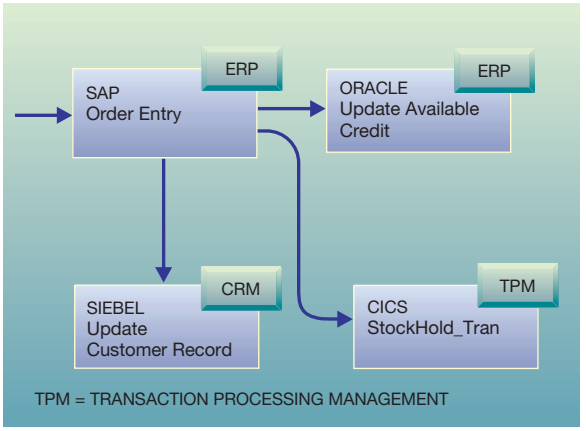
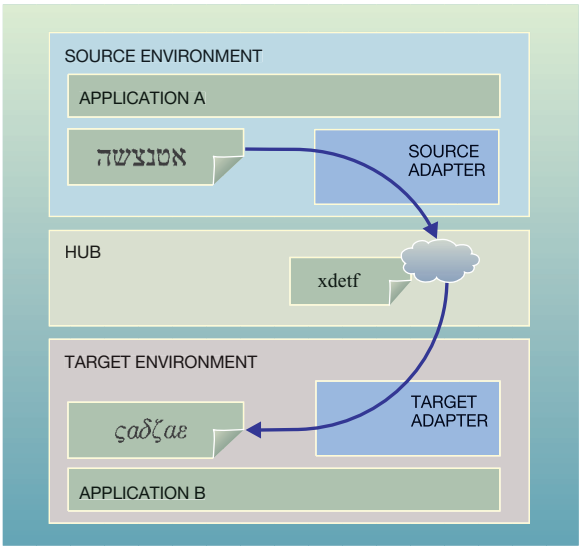


Figure 2 Source and target adapters



The hub provides a neutral format for all data interchanged between adapters. The fundamental advantage of a neutral format is that it reduces combinatorial complexity: one adapter does not need to know the data format another adapter is expecting or producing; it simply understands the specific format of its application, as well as the neutral format of the hub, and it can transform between these two formats. In some application areas, international standardization efforts specify such neutral formats (see, for example NIST[8] and Leymann[9] for neutral

Figure 3 Following corporate policies

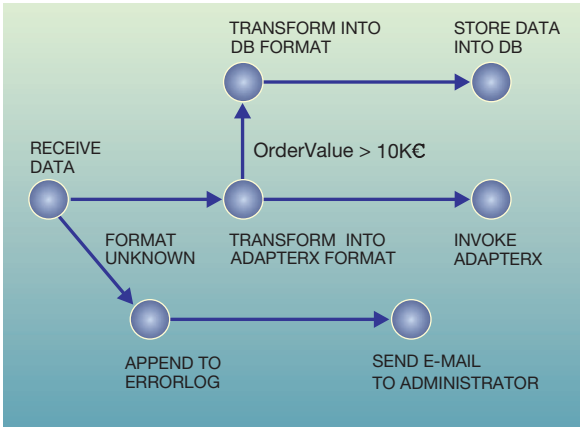
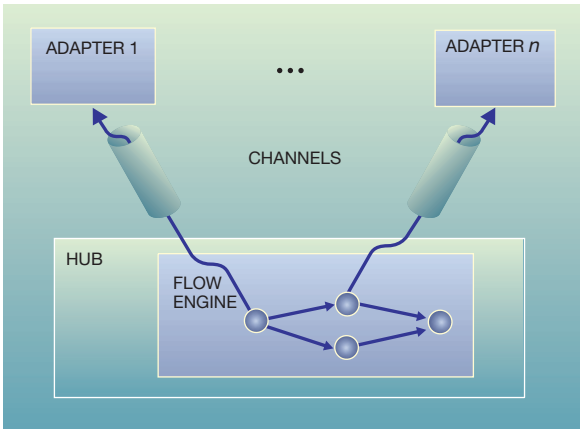


Figure 4 Abstract architecture of an EAI environment



formats of product definition data). In case of n different adapters, at most $n + n$ transformations are needed (from each application-specific format into the neutral format and *vice versa*). Without such a neutral format, each of the n adapters, in the worst case, would have to transform into the $(n - 1)$ different formats of the other adapters and *vice versa*, that is, $n \times (n - 1)$ transformations would be needed.

By introducing a hub as intermediary between adapters, additional quality of services can be achieved without additional programming in the adapters. For example, a source adapter may produce data that are of interest to a certain target adapter, but the target adapter may not be actually running. The hub may act as an active buffer, storing the data produced and passing it to the target adapter at a later time, thus increasing the availability of the overall system. Furthermore, adapters and the hub can reside in different environments and on different machines, supporting heterogeneity and scalability.

Most importantly from a business perspective, the hub can follow corporate policies when mediating data between adapters. For example (see Figure 3), when the hub receives information about an entered order, it may transform the data into the format expected by another adapter, and, if the value of the order exceeds a certain threshold, it may store the data into a database (DB) to keep track of high-value orders. In case the data format is unknown, the hub might write the data to an error log and inform an administrator, via e-mail, for corrective actions.

The directed graphs shown in the figure are called *flow models* (described in the next section). They typically represent (parts of) business processes reflecting corporate policies. Flow engines execute such a model by navigating through the graph and performing the actions specified at each node reached (see Leymann and Roller[10] for an introduction to flow technology). A hub includes a flow engine to run flow models that reflect corporate policies when mediating data between adapters. A flow model can define how to use multiple adapters of different applications within a single flow to implement the integration scenario from Figure 1, for example.

Because of the inherent heterogeneity of EAI scenarios, a hub must communicate with adapters via a multitude of formats and protocols. A particular combination of formats and protocols used to exchange data between adapters and a hub is called a *channel*. For example, a hub can use a SOAP/HTTP (Simple Object Access Protocol/HyperText Transfer Protocol) channel to communicate with one adapter, a MIME/SMTP (Multipurpose Internet Mail Extensions/Simple Mail Transfer Protocol) channel for another adapter, and a Java**/JMS (Java Message Service) channel for yet a different adapter. Figure 4 depicts the relationship among a hub, a flow engine, channels, and adapters for solving EAI problems. Note that "channel" is a generic concept and is not restricted to hub environments; in general, the communication between applications can be based on channels.

Finally, building adapters is not a trivial task. Thus, standards are needed for building adapters that can be used within each standard compliant environment. For the J2EE** (Java 2 Platform, Enterprise Edition) environment, the J2EE Connector Architecture[4] defines such a standard. An adapter built according to this standard can run in every J2EE-compliant application server and can be used by each J2EE application to access the enterprise information system corresponding to the adapter. For an environment provided by a relational database, the SQL/MED (Structured Query Language/Management of External Data) standard[2] defines a special kind of adapter called a *wrapper*. A wrapper allows an external data source to be rendered as a table and accessed directly from the database system. Wrappers have special semantics that allow them to be tied into the query processing of the database system. Hergula[11] provides a comparison between connectors and wrappers.

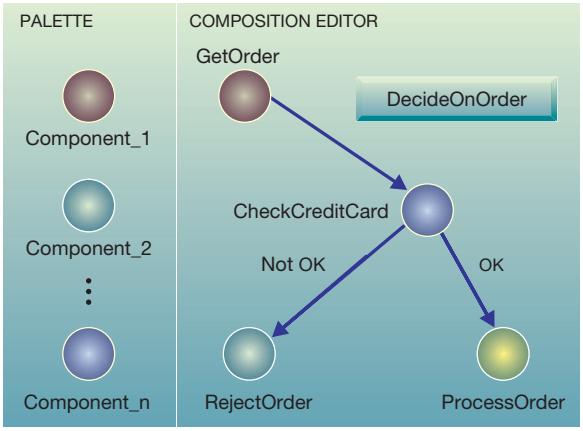
Access to applications can also be provided via *Web services*. Briefly, Web services are loosely coupled software granules available via common Internet technology.[12–14] Thus, Web services can be perceived as a base for implementing adapters, especially for supporting access over the Internet to functions provided by business partners; standardization is progressing in this area.[15–17]

Flow-based programming model

Flow technology is becoming an integral part of modern programming models.[18,19] Such models distinguish between flow logic and function logic. While function logic deals with a discrete fine-grained task (such as retrieving an order document or updating a customer record), flow logic deals with combining many functions in order to solve a more complex problem (such as processing an order). This results in a two-level programming model with programmers implementing functions (programming "in the small") and nonprogrammers implementing flows (programming "in the large").

Graphical tools may support the specification of such "flows between functions" as a graph that even nonprogrammers, for example, business specialists, can draw (see Figure 5). The nodes represent the particular functions to be invoked, and the edges of the graph represent the invocation order of the functions, as determined by business rules.[10] Within a flow composition tool, these functions may appear on a palette, then be selected and dragged into the composition editor, where they are "wired" into graphs. The types of functions span a broad spectrum, for example, stored procedures, Enterprise JavaBeans** (EJB) methods, adapters for standard applications, or Web services; note that even flows themselves can be rendered as functions and used as nodes in a flow graph. For example, the CheckCreditCard function in the example might be implemented by an EJB bean, whereas the ProcessOrder function might be a complex flow.

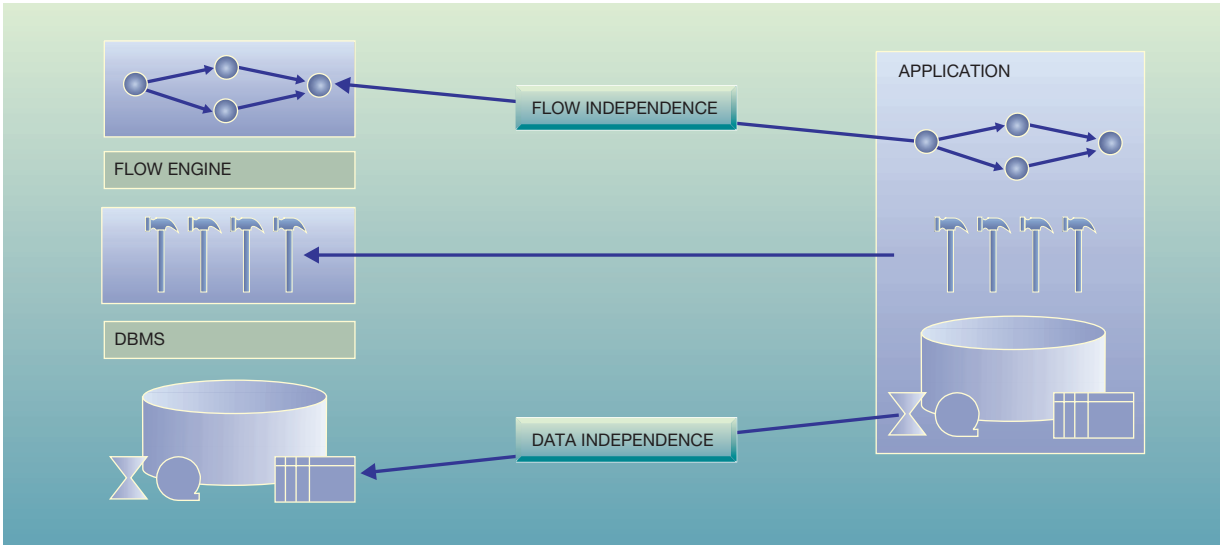
Figure 5 A flow composition tool



In general, flows are not as simple as the one shown in Figure 5, and they are often not as low-level as in Figure 3, but rather define how to process claims, how to settle trades, and so on. It is important to note that there is no special area of applicability of flow technology, just as database systems are not restricted to particular application areas. The business itself determines the business processes.

The flow-based, two-level programming model continues the separation of concerns that began with the introduction of relational database technology and results in increased flexibility of application functions built according to this model (see Figure 6). Relational database technology allows many changes of the schema of the database underlying a particular application without affecting the code of the application functions. For example, indexes might be added or dropped, tables might be moved to different devices, and so on. The application functions are said to be *data independent* in that sense. This independence was achieved by moving data management-specific logic from the application into the da-

Figure 6 Data and flow independence of applications



tabase system. Similarly, moving flow logic and composition logic from the application into the flow engine results in applications that are called *flow independent*. The wiring together of the functions can change without affecting the functions themselves, but the result may be an application with a very different behavior. The flow engine simply interprets the changed flow model and invokes the functions according to the business policy or business process it represents.

Because flows can wire together functions that are themselves flows, the flow-based, two-level programming model is recursive and can itself contribute to the flow independence of functions. Flow-independent functions typically make no assumptions about their context, that is, the order in which they are invoked. This increases the reusability of the functions.

The use of flow technology in building applications and functions introduces a higher degree of flexibility in solving business problems. Business processes or policies can be changed much easier, even by nonprogrammers. Thus, a company can react much faster to changes in the business environment.

Flow-based applications inherit a number of properties that are otherwise much harder to achieve. Such applications can be performed in a truly parallel mode; for example, the store and invoke functions in Figure 3 may run in parallel because they are on parallel paths of the graph. Flow-based applications are inherently distributed and heterogeneous, because the functions invoked by the flow engine may run on different machines and in different environments. As we will see in a later section, such applications may be recoverable according to extended transaction models.

Sample use of flows in function integration

A flow can invoke a function that is another flow, that is, flows can be nested. This allows flows to be used at different levels. For example, a flow can be used to build an adapter to get data from a standard application, that is, the flow transforms data into a format appropriate for the application programming interface (API) of the standard application, uses the API, with the transformed data as parameters, to retrieve data, and finally transforms the retrieved data into a format expected by the invoker. A flow can be used to wire more than one adapter together to get data from multiple standard applications and manipulate others, or wire together Web services that retrieve or manipulate data from various sources from the Internet.

The underlying abstract principle is *function integration*. *Local functions* $\{f_1, \ldots, f_n\}$ are provided to access otherwise opaque data sources. Each local

function f_j has input data $\{i_k\}_{k\in I}$ and might produce output data $\{o_k\}_{k\in J}$. A *global function* must access or manipulate a collection of opaque data sources in a single step. To do this, a global function uses a collection of local functions. But there are dependencies between the local functions, for example, the functions must be executed in a particular order and the output produced by one local function may be input for another. These dependencies are reflected by wiring the local functions into a flow that represents the global function. The input parameters of the global function, as well as output parameters of some local functions, provide input for some other local functions, and the output of the global function is an aggregation of the output of some of the local functions (see Figure 7). Note that this requires additional capabilities to be supported by flow engines, in particular, the ability to specify data flows (shown as dashed arrows in the figure) separate from control flows (shown as solid arrows); see Leymann and Roller[10] for the details. Finally, each such global function is again a local function, that is, it provides access to an opaque data source—the "federation" of the sources encapsulated by its encompassed local functions.

In a more concrete scenario, the local functions provide access to data managed by standard applications and legacy applications, or any other data source. The schema of the databases underlying these applications or data sources is unknown. Each local function produces output according to its signature and this signature is considered the schema of the data retrieved from the applications' databases. When data from a series of these databases is needed, a global function must be created that wires together the corresponding local functions into a flow model. A flow engine then executes the flow and invokes the local functions in the specified order. It may temporarily store the output data of the local functions and make them available to succeeding local functions in the flow, as defined by the data flow constructs. Finally, it passes the aggregated output to the output parameters of the global function, that is, the flow. In this way, the global function or flow makes the underlying collection of the data in the EIS available.

Flows in database systems

Often, the EIS data that are available via flows need to be directly manipulated in a database environment. Ideally, the data to be manipulated should be available as a table. By rendering flows as table-valued user-defined functions (UDFs) and wrappers, function integration and data integration technology is brought together.

Figure 7 Global functions are flows between local functions

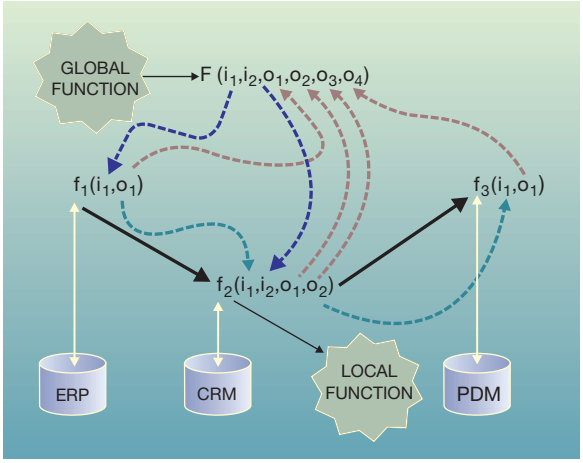


Figure 8 Flows as table-valued UDFs

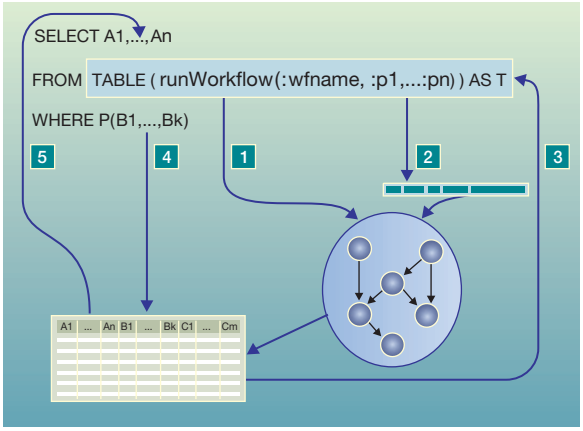
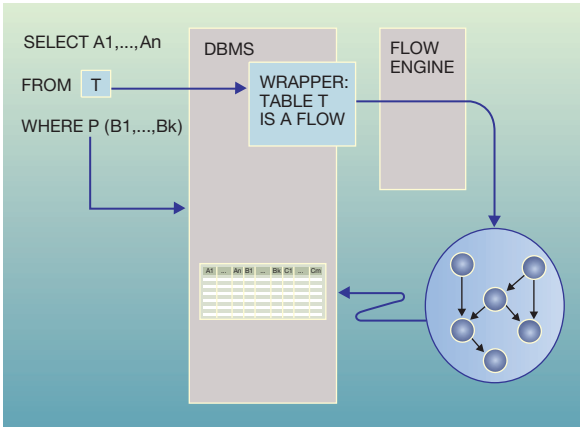


Figure 8 shows a flow that is rendered as a UDF, and its usage in a SQL FROM clause. The user-defined function runWorkflow is a façade, for the corresponding API of the flow engine, that allows an instance of a flow model to be started. The name of the flow model might be a parameter of the UDF. The (other) parameters of the UDF are those to be passed to the flow as its input parameters. In the course of its regular processing, the database system invokes the

Figure 9 Flows as wrappers



UDF, which starts the flow and receives its output. The output of the flow is a collection of tuples that can then be processed by the database system. In particular, predicates from the WHERE clause can be applied, as well as projections onto the subset of relevant columns. For more details see Wagner.[20]

In a similar manner, flows can be rendered as SQL/MED wrappers, as indicated in Figure 9. For this purpose the signature of the output of a flow model is defined as an external data source, that is, as an SQL/MED foreign table. The corresponding SQL/MED foreign data wrapper then uses the APIs of the underlying flow engine to start an instance of the flow model. This is straightforward when no parameters have to be passed as input to the flow. Flow engines are very simple SQL/MED servers and typically do not understand relational algebra operators. In order to pass input parameter to a flow, a detour must be made through function mappings.[20]

The invocation of flows from within a database system has an interesting similarity to stored procedures: flows can be seen as *federated stored procedures*,[21] that is, as a generalization of stored procedures for federated database systems. The argument in favor of federated stored procedures is exactly the same as for regular stored procedures (see Figure 10). An application that manipulates a collection of data sources available under a federated database system runs a "script." The script invokes functions that request, via the federated database system, manipulations of the data sources. Each request involves communication between the application and the federated database system.

Typically, it is much more efficient to run the complete script within the federated database system. This reduces the number of interactions between the application and the database system to one, avoiding the exchange, and the possibility of wire tapping, of intermediate data between the functions of the script. The application simply calls the federated stored procedure, then the database system invokes the corresponding flow within the associated flow engine and returns the result of the flow to the application. It is the flow engine that invokes the functions manipulating the data sources, resulting in additional benefits because functions can be used (e.g., adapters) that have not been built for invocation by a database system, in contrast to UDFs and wrappers.

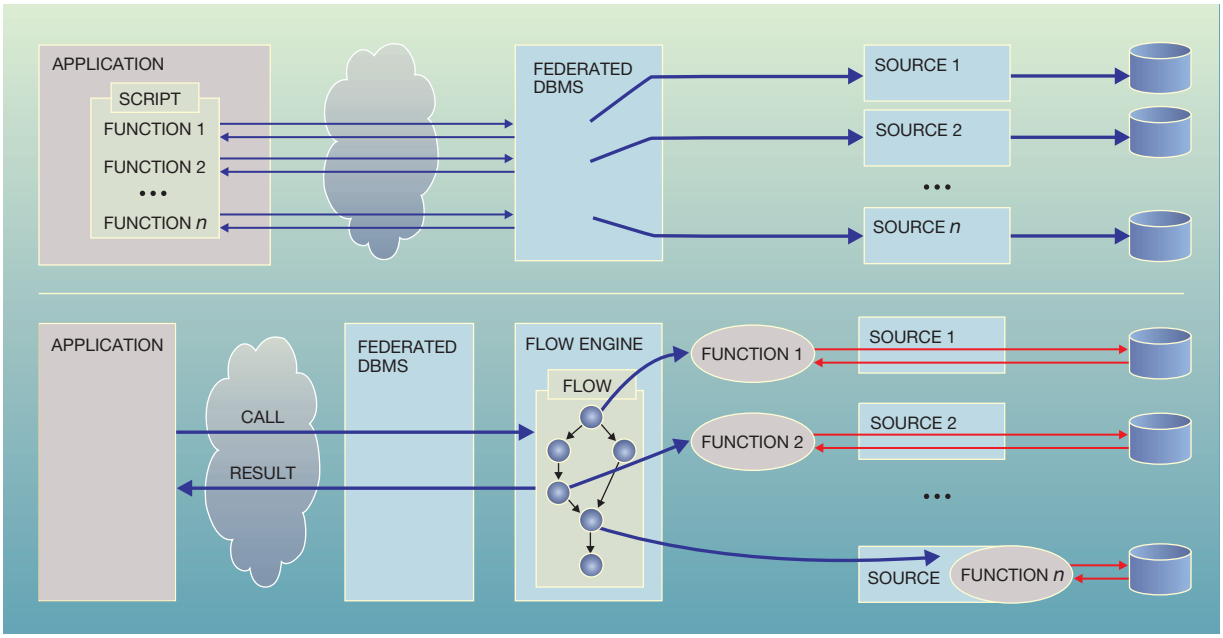
Transactions in flows

When a flow is invoked from within a database management system (DBMS) via one of the mechanisms discussed in the previous section, the invocation of the flow may be just one operation within a unit of work. The invocation of a flow might be mixed with regular SQL statements, and the effects of either all or none of the operations must be reflected in the underlying data stores. Thus, a flow must participate in the unit-of-work processing of the database system. This can be achieved by various means.

Flows can bound functions that are implemented as transactions into a new transaction. For example, the flow in Figure 11 (a slight variation of the flow from Figure 3) calls the method of an EJB bean and (under the specified condition) stores the data received in a database. The EJB bean is running under transaction protection and the storage operation is also a transaction. When both functions are executed in a flow either they must both complete successfully, or neither is performed; that is, both functions must be executed by the flow engine as a single transaction.

A group of transactions bound together within a flow is referred to as an *atomic sphere*. Whenever the flow engine enters an atomic sphere, it begins a new transaction, and each contained transaction is invoked under the atomic sphere transaction context. If any contained transaction fails, the atomic sphere fails and is automatically retried until the number of tries reaches a threshold. If all contained transactions complete successfully, the atomic sphere is considered to have completed successfully and the flow continues as usual. See Leymann and Roller[10] for more

Figure 10 Federated stored procedure



technical details and the precise definition of, and assumptions behind, atomic spheres.

Flow engines also allow collections of functions that are not themselves implemented as transactions to be grouped into transactions. For example, one function within a flow writes a scanned-in job application letter to a file, and another function invokes a nontransactional interface of a standard human resources application to insert the name and address of the applicant. Both functions must be performed successfully, or the effect of one function must be undone if the other function fails. For this purpose, each of the functions is associated with a "compensation function," that is, a function that undoes what the intended function has done. The compensation function for writing the scanned-in letter to a file discards the file, and the compensation function for inserting the applicant information deletes this information. The flow engine will perform the compensation function associated with an intended function whenever the intended function has been run and its coupled function has failed.

A group of functions within a flow that must jointly run successfully or be undone by compensation is called a *compensation sphere*.[22] Each function within a compensation sphere has a compensation function associated with it. For example, in Figure 12 compensation sphere S groups together the functions B, C, D, and E; the compensation function of B is called ^{c}B and so on. When B, C, and D have been performed and E fails, the flow engine will execute the compensation functions ^{c}D, ^{c}C, and ^{c}B to undo the effects of B, C, and D.[1,22] For background on trans-

Figure 11 An atomic sphere

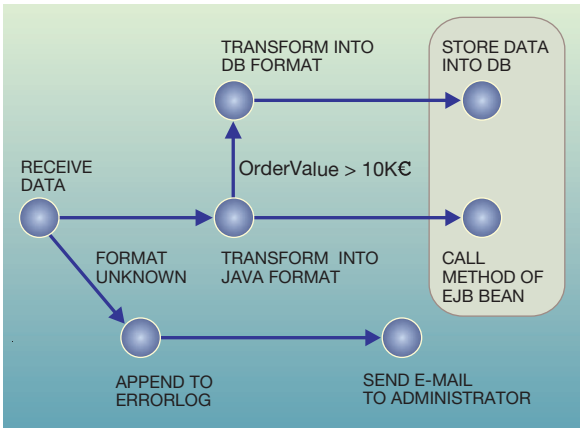


Figure 12 Compensation spheres

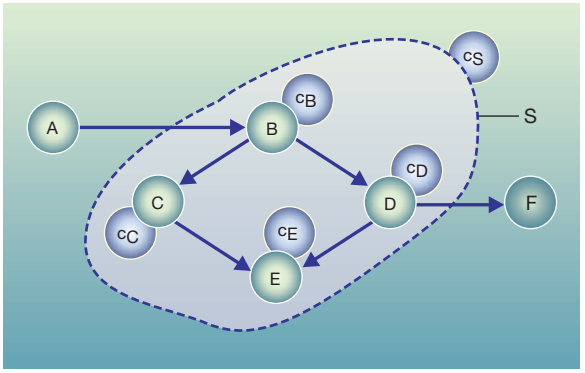
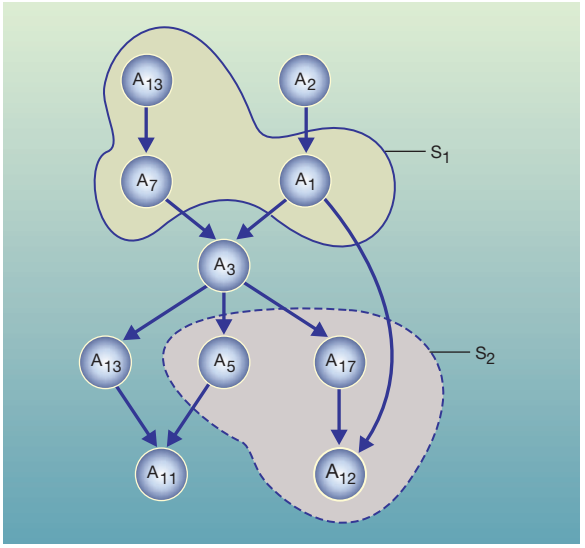


Figure 13 Spheres in flows



action concepts and an in-depth treatment of related transaction technology see Gray and Reuter[23] and Weikum and Vossen.[24]

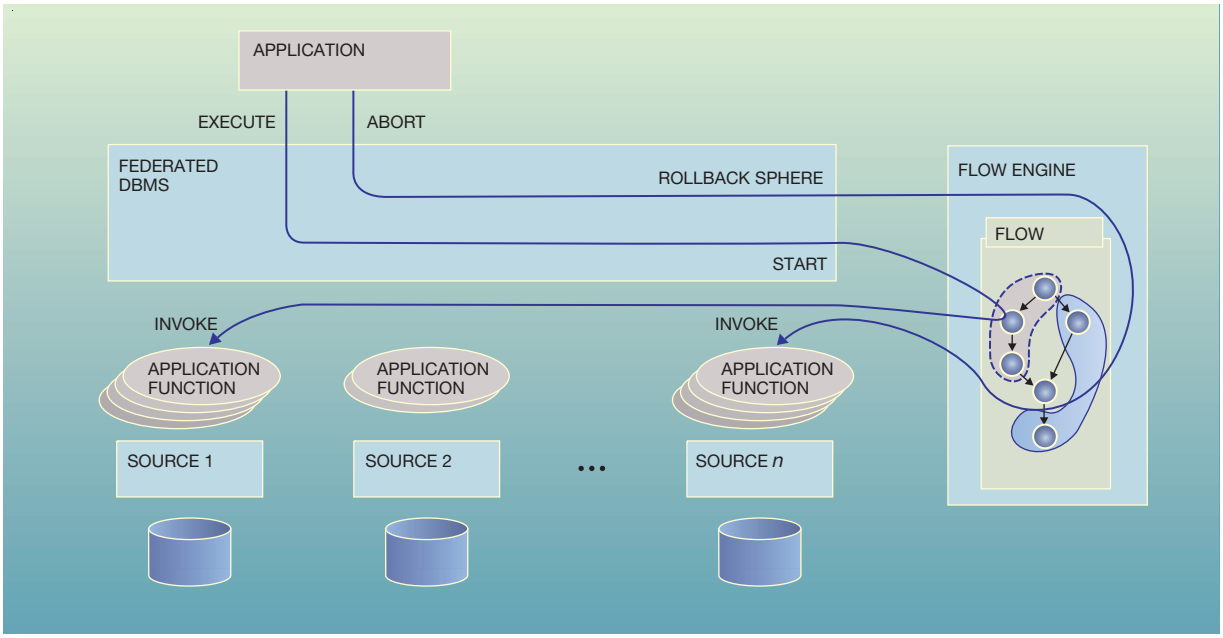
The use of spheres to collect functions with particular properties within a flow allows us to specify a broad spectrum of behavior. Not only transactional behavior, such as atomic spheres and compensation spheres, can be defined, but also more general behavior, such as joint fault handling or time-out processing, may be specified. Spheres can also have general properties, such as being *permeable* or *impermeable* (indicated by a dotted line or a solid line in Figure 13). A permeable sphere maintains its behavior after the flow leaves the sphere, whereas impermeable spheres do not. A compensation sphere might be permeable, that is, the compensation of the functions performed within a sphere might be requested long after the intended functions in the compensation sphere were completed. An atomic sphere is impermeable, that is, once the flow has moved on, its actions cannot be undone (otherwise, the usual implementation of ACID [atomicity, consistency, isolation, and durability] semantics would require locks to be held for an indefinite duration).

Another general property of a sphere is whether or not it must be *interruptible*. When a sphere is interruptible, the flow engine stores state information before and after each function within the sphere is run. In case of a failure of the environment, the flow itself continues, after restart, where it left off.[10] A sphere that is noninterruptible does not have restart capabilities, but it typically runs orders of magnitudes faster. (The noninterruptible parts of a flow are also referred to as *microflows*.) Thus, the appropriate properties to specify for sphere performance must be considered along with the overall quality of services.

In summary, the use of atomic and compensation spheres helps in controlling the backward recovery properties of (parts of) a flow. The notion of interruptability helps in controlling forward recovery or Phoenix behavior.[23] Thus, flows allow us to control fundamental transactional aspects of groups of functions.

When flows are made available in a database system, their transactional capabilities can be exploited there. Figure 14 shows a database system that cooperates with a flow engine via techniques similar to the ones described earlier. Thus, the application can access or manipulate external data sources via appropriate flows started by the database system in the associated flow engine. When the database system or the application itself detects an error and has to abort, the database system may request the flow engine to undo the involved spheres; the detailed discussion of the underlying mechanics is outside the scope of this paper. But it should be obvious that by coupling flow technology with database technology, the management of long-running transactions becomes available to database applications. Note that this does not solve the issues around traditional transactions in federated databases such as global serializability.[24]

Figure 14 Managing extended transactions via flows



Conclusion

In this paper we give a high-level overview on adapter technology and its relationship to flow technology. We sketch how both technologies work hand-in-hand to solve many aspects of the enterprise application integration (EAI) problem. In particular, retrieving data from and manipulating data in external data sources can be achieved in a very flexible manner. The flexibility is a result of the flow-based programming model, based on flow technology, that is becoming more and more accepted. This programming model helps us to solve fundamental aspects of the function integration problem.

We discuss various ways to couple a database system and a flow engine in order to make function integration capabilities available to database applications and database users. Especially important, advanced transaction features supported by flow engines may become part of a database system. Together, the combination of a database system and a flow engine are a step forward in providing advanced information integration capabilities.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references

1. A. P. Sheth and J. A. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases," *ACM Computing Surveys* **22**, No. 3, 183–236 (1990).
2. *Information Technology—Database Language—SQL—Part 9: Management of External Data (SQL/MED)*, ISO/IEC 9075-9, International Standards Organization (2001).
3. K. Hergula and T. Härder, "A Middleware Approach for Combining Heterogeneous Data Sources—Integration of Generic Query and Predefined Function Access," *Proceedings, 1st International Conference on Web Information Systems Engineering*, Hong Kong (June 19–21, 2000), pp. 26–33.
4. *Java 2 Enterprise Edition, J2EE Connector Architecture Specification*, Version 1.0, Sun Microsystems (2000).
5. See http://www.rosettanet.org.
6. See http://www.sap.com/solutions/compsoft/scenarios/te/docs/ca_edi_testplan_40_en.pdf.
7. See http://www.sap.com/solutions/compsoft/scenarios/te/docs/rfc_description.pdf.
8. See http://www.nist.gov/sc4/www/stepdocs.htm.
9. F. Leymann, "Towards the STEP Neutral Repository," *Computer Standards and Interfaces* **16**, No. 3, 299–319 (1994).
10. F. Leymann and D. Roller, *Production Workflow*, Prentice Hall, Inc., Upper Saddle River, NJ (2000).
11. K. Hergula, "Wrapper und Konnectoren—geht die Rechnung auf?" *Proceedings, Datenbanksysteme in Büro, Technik und Wissenschaft*, Oldenburg, Germany (March 7–9, 2001), pp. 461–466.
12. See http://www-4.ibm.com/software/solutions/webservices/resources.html.
13. S. Burbeck, The Tao of e-Business Services, available at

http://www-4.ibm.com/software/developer/library/ws-tao/index.html.
14. D. Ferguson, IBM Web Services: Technical and Product Architecture Roadmap, available at http://www-4.ibm.com/software/solutions/webservices/pdf/roadmap.pdf.
15. See http://www.w3.org/TR/SOAP12.
16. See http://www.w3.org/TR/wsdl.html.
17. See http://www.uddi.org.
18. F. Leymann and D. Roller, "Workflow-Based Applications," *IBM Systems Journal* **36**, No. 1, 102–123 (1997).
19. G. Wiederhold, P. Wegner, and S. Ceri, "Towards Megaprogramming: A Paradigm for Component-Based Programming," *Communications of the ACM* **35**, No. 2, 89–99 (1992).
20. R. Wagner, *Integration of Workflows into a Federated DBS with SQL/MED*, master's thesis, Institute of Computer Science, University of Stuttgart, Germany (2001).
21. F. Leymann, "A Practitioner's Approach to Database Federation," *Proceedings, 4th Workshop on Federated Databases—Integration of Heterogeneous Information Sources*, Berlin, Germany (November 25–26, 1999).
22. F. Leymann, "Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems," *Proceedings, Datenbanksysteme in Büro, Tecknik und Wissenschaft*, Dresden, Germany (March 22–24, 1995), pp. 51–70.
23. J. Gray and A. Reuter, *Transaction Processing*, Morgan Kaufmann Publishers, San Mateo, CA (1993).
24. G. Weikum and G. Vossen, *Transactional Information Systems*, Academic Press, San Diego, CA (2002).

Accepted for publication July 15, 2002.

Frank Leymann *IBM Software Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (ley1@de.ibm.com).* Dr. Leymann is an IBM Distinguished Engineer, a member of the IBM Academy of Technology, and a professor of computer science at the University of Stuttgart, Germany. He is the chief architect of IBM's workflow technology, and a member of the AIM Architecture Board that sets the overall technical direction for IBM's middleware products. Before his current position he worked on database systems, database tools, and transaction processing. Dr. Leymann has published many papers in various journals and conference proceedings and filed a number of patents, and he is the coauthor of textbooks on repositories and on workflow systems. He has served as a member of program committees and organization committees for many international conferences and is editor of the journal of the DBMS special interest group of the German computer society (GI).

Dieter Roller *IBM Software Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (rol@de.ibm.com).* Mr. Roller is an IBM Senior Technical Staff Member and a member of the IBM Academy of Technology. He has held several technical and management positions during his IBM career. His current focus is on the architecture and design of IBM's MQSeries® Workflow product, contributing to all facets of the development and enterprise-wide deployment of flow-based applications, and he is deeply involved in customer projects in this area. Mr. Roller has published papers in various journals and conference proceedings, mainly on workflow technology, filed many patents, and given talks at conferences and professional society meetings. He is the coauthor of a textbook about workflow systems.