Adaptation of Software Entities for Synchronous Exogenous Coordination

An Initial Approach

Nikolay Diakov and Farhad Arbab

Centrum voor Wiskunde en Informatica,
P.O. Box 94079, 1090 GB Amsterdam,
The Netherlands,
{nikolay.diakov, farhad.arbab}@cwi.nl

Abstract. In this paper we present an ongoing work on a framework for adaptation of heterogeneous software entities to allow their integration together with the help of *synchronous connectors*. By using synchronous connectors for software integration, we intend to make it possible to significantly reduce the time and money spent for programming fortifications against unwanted behavior, as compared to the time and money spent for programming *explicit* business scenarios. In this paper, we describe our initial approach to how one can adapt a large class of existing software entities that offer standard RPC-style operational interfaces, for integration through an arbitrary synchronous Reo connector.

1 Introduction

A business scenario *explicitly* describes a recipe for performing some necessary steps that ultimately lead to the production of a desired end-result. Take as an example a holiday reservation service that requires the reservation of a flight, a hotel, and a car. A successful reservation requires all of the individual steps to happen and it does not require them done in any particular order.

A business scenarios typically states what should happen, which at the same time means that anything omitted from the scenario that may prevent the achievement of any explicitly mentioned results, should not happen. We distinguish two general methods that business automation developers use to build a software system that enforces a business scenario: (1) *direct* – through programming the steps that the scenario says should happen, and (2) *fortification* – through programming to prevent the unwanted behavior that a scenario does not mention explicitly, but common sense and experience dictate should not happen in order to make sure that the software systems always follows the scenario that it automates.

In our experience, developers often spend less time for designing and programming business scenarios directly, than for designing and programming *fortification code* for these scenarios. Developers spend even more time for debugging the manually-developed fortification code, often designated to operate in a

large and dynamic distributed environment. The high cost of enforcing explicit behavior of software applications by default, through developing fortification code, sometimes even becomes prohibitive. We consider this as one of the main problems in contemporary business automation. Therefore, any tool or technique that improves on the situation has the potential to generate serious value for the software development industry.

Large and dynamic distributed systems, such as the Internet, offer great potential for business automation to take advantage of. At the core of programming in a distributed environment lies the capability and the necessity to coordinate independent activities together, to achieve a common goal. Using *connectors* to directly and exogenously (i.e., from the outside) coordinate the activities performed by independent, autonomous, and possibly physically distributes software entities, has become a promising technique for integrating heterogeneous software in large, dynamic and distributed computing environments. Channel-based coordination languages, such as Reo [1], facilitate the modeling, construction, and execution of such connectors.

The database community has studied the problem of directly enforcing behavior in an automatic manner by introducing the notion of a *transaction*. Among other things, any steps undertaken by design within the context of a transaction, appear to an entity outside of this context as one *atomic* activity. Most databases support transactions by automatically taking care of any clean up necessary after an incomplete transaction, thus enforcing only explicit (completely successful transaction representing a) business behavior. Packages for doing business transactions have also appeared (MTS [2], IBM CICS [3]) that operate at the level of inter-component interactions. These packages allow developing components that can participate in transactions. Current vendor technologies: (a) focus on providing transactions for particular narrow domains of applications (e.g., databases); (b) can provide a wider choice of applications but at the cost of leaving too much for the application developers to do themselves (e.g., transactions cover only basic sequences of simple component interactions); and (c) do not provide sufficient support for composition and nesting of existing transactional components. Synchronous programming languages also allow enforcement of synchronous behavior. ESTEREL [4] and LUSTRE [5] offer a practical approach and have large commercial acceptance especially in the embedded systems domain. This class of languages, however, enforce a click-step style of synchrony through all components in the system – something inappropriate in a distributed system, in which individual components may want to execute at their own speeds.

The rest of this paper has the following structure. We elaborate on the notion of enforcing explicit business scenarios in Section 2, and introduce our approach to specify and enforce them using Reo. In Section 3, we analyze the issues related to our goals, and we present what designers need to do to adapt a COTS component built using common middleware technology. We present a proof-of-concept implementation of a Simple Transactional API for the MoCha coordination middleware in Section 4, using a special $f(x)$ channel, allowing us to adapt an example database application for use with synchronous connectors.

2 Enforcing business scenarios

In a business scenario, to achieve the end result one usually performs all necessary steps of the scenario and therefore no intermediate result can count as a success. Referring to individual business scenarios as inherently *atomic* seems natural – the term business transaction predominates the language we use when communicating about business. In this paper we disregard long running business interactions, which may tolerate partial failure by allowing for *compensation* activities. Technically, we do not consider these as atomic in the classical sense introduced with ACID transactions (where A stands for atomicity) [6].

Automating a business scenario in a distributed environment requires the coordination of the behavior of several otherwise independent software components. After some refinement of a business scenario, software designers typically come up with some protocol to coordinate the activities of the instances of the necessary software components to achieve the desired result. In a component-based system, a coordination protocol resides within the "glue code" that composes some (possibly independent, e.g., off-the-shelf together as well as home-grown) components. *Exogenous* coordination treats glue code as a first-class modeling entity that resides outside of any of the components it coordinates (hence "exogenous"). Exogenous coordination promotes loose-coupling between components, which in turn improves software reusability, maintainability, change management, and with proper technological support, allows dynamic re-configuration [7, 8].

Our work focuses on facilitating a component-based software development process that allows and encourages the direct (semi)*automatic* enforcement of explicit behavior by means of exogenous and *synchronous* coordination, as opposed to *manual* fortification against unwanted behavior by means of additional developers work. We aim to allow integration of commercial off-the-shelf (COTS) components into atomic implementations of business scenarios (transactions). Furthermore, we aim at facilitating composition of existing transactions. To specify and implement exogenous and synchronous connectors, we use the Reo coordination language [1]. Reo offers both synchronous and asynchronous coordination primitives, called channels.

The concept of synchrony in Reo directly relates to the notion of atomicity we introduced earlier. Consider the synchronous channel `Sync`. A `Sync` has two channel ends, an input and an output. A request for writing a data item on the input end of a `Sync` succeeds if and only if a pending request exists to take a data item on the output end of the channel. Since neither a write nor a take request can succeed on its own, `Sync` appears to combine the acts of writing and taking of data items into a composite atomic act enforced by this channel. Composing more synchronous channels together using Reo's topological operations allows one to create a connector that makes an arbitrary number of write and take activities appear atomic, with the side effect of transporting data items, e.g., across some communication infrastructure.

The Reo coordination language by design supports compositionality [9], allowing compositional construction of complex applications [10]. Compositional-

ity in Reo permits nesting of synchronous connectors for free – behaviorally, we interpret the notion of nesting as composition of constituent behaviors to form a new higher order behavior. Reo also comes with the added value of the ability to compose during runtime. Consider an example in which an electronic auction system supports many participants: an auctioneer hosts the auction, and an owner determines the initial conditions of the auction [11]. Now suppose that several of the participants dynamically enter into an alliance in order to improve their outcome. An auction protocol and an alliance protocol implemented in Reo, allow composition just by using some auxiliary synchronous channels to connect the alliance to the auction as a new participant [12]. From the auctions perspective, this looks like nesting an alliance behavior within a participant behavior to allow participation in the an auction (transparent to the auction).

To facilitate the integration/assembly of COTS component with synchronous connectors specified in Reo, we need to provide the necessary minimum technology that enables this integration. We cannot assume that a COTS component that provides the necessary functionality comes also with support for participation in a transaction. For example, if a transaction does not succeed, a component may need to restore its previous internal state in order to remove the traces of any intermediate results of the unsuccessful transaction; something the original designers of a component may not have intended it to do.

To summarize our initial approach: we analyze what facilities we need to provide so that a designer can *adapt* a COTS component with little effort for integration with the Reo technology; we implement the identified facilities in the MoCha middleware [13] – an initial implementation of Reo; we then provide an example of the use of these facilities to demonstrate the feasibility of our proposition.

3 Middleware for synchronous interactions

In this section, we analyze our problem from several perspectives. We explore what it means for a component to interact synchronously (as we defined it). We summarize the interaction patterns of common component middleware. Finally, we present the current state of coordination middleware that we can use.

3.1 Synchronous interactions

We assume that a coordination middleware that "speaks" Reo enforces the necessary synchronous coordination among arbitrary individual COTS components. To integrate a particular component technology with such a coordination middleware, we need to (a) allow a component instance to interact with a synchronous connector technologically, and (b) since, in the general case, we cannot assume that a COTS component supports transactions, we may need to adapt it to support them. We consider (a) as a matter of proper wrapping, done once at the technological level for a particular component middleware, and therefore

we do not discuss it in detail. To do (b), however, one may need additional work per individual component depending on what it does.

Designers can easily integrate *stateless* components into a transaction. In stateless components, every interaction depends entirely on its immediate parameters and a component instance does not keep any information about its past interactions in the form of some internal state. For example, a component that sorts an array passed to it in a parameter and returns the result does not need to preserve a state. A *stateful* component, on the other hand, requires certain adaptation to allow it to clean up its internal state, if a transaction in which it participates fails. The Reo computational model [14, 15] uses a protocol similar to the well-known two-phase commit protocol (2PC) to implement its synchronous connectors. Adapting stateful components for the 2PC provides one solution to their integration: the connector plays the role of the global coordinator in the 2PC. Depending on the actual component middleware, in addition to state, designers may also have to take care of various concurrency issues, such as call isolation among concurrent access sessions, re-entrance within the same session, object activation/deactivation, persistence, and others, which we do not discuss in this paper.

3.2 Component middleware

Most technologies for component-based development use an RPC-capable communication infrastructure for interaction – DCOM [16], CORBA CCM [17], EJB with Java RMI [18], and so on. A component offers its behavior in terms of an interface: a collection of individual *operations* (also called methods, functions, or procedures) specified by a signature, perhaps pre and post conditions per operation, and some relations among the operations (e.g., order of calls, etc). Thus, we do not much limit our options by considering a component to represent an RPC-enabled library of (a) functional blocks defined in formal operational interfaces, and (b) *software protocols* for using them, often specified informally. In this paper we deal with facilitating (a). We leave (b) for future work.

The RPC protocol for invoking operations blocks the *caller* until a result becomes ready. In the blocking 2PC protocol, the component that serves an RPC call (*callee*) also blocks until the transaction in which it participates (through interactions with a synchronous connector) completes, either failing or succeeding. A designer can hide this blocking from a component through a generic wrapper that mediates all interactions with the component. This wrapper exposes the necessary facilities for notification of success or failure through a generic programming interface. Integrators use this interface to adapt their components for synchronous integration.

3.3 Coordination middleware

The MoCha middleware [13] constitutes the current initial implementation of the Reo coordination language. MoCha implements individual synchronous

channels, which also support mobility. The individual functional blocks, the operations, that a component offers through its operational interface take an input (parameters) and produce an output (result and/or exceptions, error codes, etc). In this sense, an operation behaves somewhat similar to a channel in Reo. It seems natural then to view a component instance as a collection of channel instances. For synchronous connectors, we require operations to appear as synchronous channels. Thus, from the point of view of a connector designer, we regard an operation as a `Sync` channel that synchronizes its two ends and transports data. This data transport, however, has the side effect of computing some (for stateful components – possibly history sensitive) function $y = f(x)$ on that data (here x stands for a tuple of input parameters and y represents a tuple of output parameters or results). From the point of view of an application integrator who needs to adapt a component, to appear as a synchronous channel, the code of each (stateful) operation should become transactional. In MoCha, an Simple Transactional API (STAPI) for 2PC-style, offered next to the standard API for implementing new channels, can aid the integrator in the adaptation process.

4 Proof of concept

In this section we assume the role of an integrator who needs to adapt a simple database access component for use with synchronous connectors for the purpose of *logging of transaction successes* (e.g., to use for auditing). Naturally, the logging of a success should become a part of the transaction itself (through synchronous integration, logging success only when the transaction completes. Since we do not have a complete Reo implementation, we use the Java-based MoCha middleware, which provides only basic synchronous connectors called synchronous channels.

We build a specific general `f(x)Sync` channel, which internally provides a STAPI for MoCha (STAPI4MoCha) programming interface. Integrators can use this interface to allow something (a connector) that appears as a 2PC protocol coordinator, to interact with the implementation of the component operation. The 2PC protocol requires processing of several messages: a `global prepare` sent from the transaction coordinator (in our case, the synchronous connector) to all participants, to which they respond with either `local success` or `local failure`; a `global abort`; and a `global success`. Our interface offers only two kinds of messages to the component: `global success` and `global failure`. Internally, we consider the actual invocation of an operation as the `global prepare` message, returning normal result as a `local success`, and returning with exception as `local failure`. A call to `transaction(callback interface)` within an operation establishes the callback method, to which the channel implementation will deliver the messages.

For our example, we use a simple database access component that provides access to a file storage. As a component model we assume a single Java class. We intend for the operations of reading from (given offset and size) and writing

to (given offset and data of some size) a file, to appear as operations on two respective `f(x)Sync` channels. Note that we have chosen a stateful component (the write operation), because the file represents the state kept between individual calls. In this application, we buffer the written data, preparing it for writing exclusively to the file, checking space limitations, and so on. When a `global success` arrives we flush the buffer onto the file.

5 Conclusions and discussion

We presented an approach to synchronous software integration, in which we propose to use Reo as the specification and implementation language for synchronous connectors. As an added value, using Reo as a specification language enables one to take advantage of Reo's formal semantics [19] for tool-based verification, simulation, and reasoning about software compositions. Used as an implementation language, Reo's computational model [14, 15] can enforce coordination protocols in a de-centralized, scalable, and (partially) fault tolerant manner.

In our approach, for practical reasons we have decided to focus on a basic behavior block widely used by the industry to offer behavior through remote interfaces – the RPC-style of operation invocation. By modeling an operation as a synchronous channel, we enable native integration with Reo-connectors of PRC-style operational interfaces. Providing native support for other styles of interaction, such as asynchronous message passing, message queueing and event-based notification, remains an open issue. Nevertheless, if we have a library for these interaction styles, implemented with RPC-style interfaces, we can still offer a technological solution with our framework. We see such solution as non-native, but one on top of the existing framework presented in this paper. For this kind of solutions, however, we need to have a better specification of coordination among the uses of the individual operations in an interface (provided by a component or library).

We realize that developers often describe the software protocols for using interface operations of a component in an informal language (manuals and documentations). This practice inherently has a huge potential for producing errors in the way integrators use a component. Components and libraries for high-level distributed communication protocols, such as the alternatives to RPC mentioned above, include subtle details of how multiple parties can concurrently use a component. We intend to investigate whether one can use "local" connectors that directly express and enforce any intra-component coordination among the operation calls on a component interface.

As part of future work, when the middleware that supports the full Reo becomes available, we plan to enhance it with facilities for integration with at least one common component model, such as CORBA CCM, EJB, or COM+/.Net Components.

Acknowledgements

We thank Dr. David Clarke for his comments on this work.

References

1. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science **14** (2004) 329–366
2. Microsoft Corporation: Microsoft Transaction Server (2005) http://www.windowsitlibrary.com/Documents /Book.cfm?DocumentID=405.
3. IBM Corporation: IBM CICS Transaction Server (2005) http://www-306.ibm.com/software/htp /cics/tserver/v31/.
4. Berry, G.: The Foundations of Esterel. MIT Press (2000)
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL. (1987) 178–188
6. ISO/IEC: ISO/IEC 10026-1:1992 Section 4 (1992)
7. Arbab, F.: A behavioral model for composition of software components. L'Objet (2005) to appear in 2006.
8. Arbab, F.: What do you mean, coordination? (Bulletin of the Dutch Association for Theoretical Computer Science (NVTI))
9. Arbab, F.: Abstract Behavior Types: A foundation model for components and their composition. Science of Computer Programming **55** (2005) 3–52 extended version.
10. Diakov, N., Arbab, F.: Compositional construction of web services using Reo. In Bevinakoppa, S., Hu, J., eds.: Proceedings of The second International Workshop on Web Services: Modeling, Architecture and Infrastructure, WSMAI'2004, INSTICC Press, Portugal (2004) 49–58
11. Zlatev, Z., Diakov, N., Pokraev, S.: Construction of negotiation protocols for e-commerce applications. ACM SIGecom Exchanges (2004) 11–22
12. Diakov, N., Zlatev, Z., Pokraev, S.: Composition of negotiation protocols for e-commerce applications. In Cheung, W., Hsu, J., eds.: The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, IEEE Computer Society (2005) 418–423
13. Arbab, F., de Boer, F.S., Scholten, J.G., Bonsangue, M.M.: Mocha: A middleware based on mobile channels. In: COMPSAC, IEEE Computer Society (2002) 667–673
14. Everaars, K., Costa, D., Diakov, N., Arbab, F.: A distributed implementation of Reo connectors – ongoing work at CWI (2005)
15. Costa, D., Clarke, D., Arbab., F.: Connector colouring: Towards implementable semantics for Reo – ongoing work at CWI (2005)
16. Kirtland, M.: Object-Oriented Software Development Made Simple with COM+ Runtime Services. Microsoft Systems Journal (1997) http://www.microsoft.com/msj/1197/complus.aspx.
17. OMG: CORBA Component Model Specification (2001) http://www.omg.org/cgi-bin/doc?ptc/2001-11-03.
18. SUN Microsystems: Enterprise Java Beans Specification (2002) http://java.sun.com/products/ejb/docs.html.
19. Arbab, F., Rutten, J.: A coinductive calculus of component connectors. In M. Wirsing, D.P., Hennicker, R., eds.: Recent Trends in Algebraic Development Techniques, Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT 2002). Volume 2755 of Lecture Notes in Computer Science., Springer-Verlag (2003) 35–56 http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0216.pdf.