Florian Matthes

Mobile Processes in Cooperative Information Systems

Cooperative information systems can be understood as distributed systems where multiple autonomous agents (possibly in different organizational units) have to coordinate their long-term activities towards the fulfillment of a cooperative task [DDJ+97]. For example, within an insurance enterprise the central contract management division (using a mainframe information system), a sales representative (using a laptop-based information system) and a call center (using another switch management information system) could all be involved in a single cooperative, long-term process , the contract negotiation with a particular customer.

In this paper we present an approach to the construction of cooperative information systems based on first-class persistent and mobile threads which communicate via structured *Business Conversations.*

1. Background and rationale of this work

The main task of languages and middleware services for modern information systems is to allow system builders to abstract from the details of

1. cooperation over **time**: Agents and artifacts should exist as long as required by the business processes they support, independently of the underlying language and system concepts.
2. cooperation within **space**: Agents and artifacts should be able to migrate freely within a physically distributed environment, independently of the particular system platforms or organizational structures involved in the cooperative work.
3. cooperation in multiple **modalities**: Artifacts should be accessible uniformly for agents that cooperate in different modalities like simple overnight batch processing, online transaction processing, direct manipulation by human agents via form-based or graphical user interfaces, computer-supported cooperative work by humans, computer-assisted workflow management, or automatic information processing by mobile software agents.

Our research work in the field of persistent object systems [Matt93] has focused on the issues (1) and (2) by providing full *persistence* and *mobility* abstraction in the platform-independent Tycoon system [MaSc94,MMS95b] culminating in the development of persistent migrating threads [MMS96b] which can be used to implement software agents similar to Telescript agents [Math96]. In [Matt97a] we introduced Business Conversations as a software model to address the third issue, cooperation in multiple modalities. In this paper, we put these two developments into perspective by describing the implementation of Business Conversation in Tycoon.

2. Why distributed persistent objects are not enough

One can distinguish roughly the following three (complementary) views on information systems:

Data-centered modeling: Which data structures are maintained by the system? What are their attributes and relationships? Which integrity constraints exist on these data structures?

Object-centered modeling: Which operations can be applied to these data structures? What are legal state transitions on these objects?

Activity-centered modeling: How does the system interact with its environment over time? Are there subsystems with restricted communication links to other subsystems? What are protocols between subsystems? What is the long-term goal to be achieved by a sequence of communication steps between subsystems?

These views give rise to three basic approaches to achieve cooperation between information systems which we will discuss in turn.

The cooperation of multiple applications based on distributed and persistent **business data**, for example, using relational database systems and remote database access protocols leads to client/server systems organized around centralized data servers, as exemplified by integrated business application systems like SAP R/3, Baan or Oracle Financials.

The promise of distributed object management is to arrive at more flexible, scaleable and maintainable system architectures by building cooperative information systems using distributed and persistent **business objects** [OHE96]. A business object encapsulates business data and achieves a higher degree of autonomy by restricting access to the business data through well-defined method interfaces.

Distributed business objects are a promising software structuring concept for rather tightly integrated business applications, e.g. in-house desktop clients accessing corporate business object servers. However, we believe that it does not scale well for more advanced patterns of cooperative work involving truly autonomous profit centers within an organization or involving several departments of independent enterprises which are unlikely to agree on a common business object model and a shared object infrastructure which may be expensive to maintain.

Another difficulty of this model is the fact that the interaction between business objects and the coordination of their behavior is hard-coded and often distributed in a complex way over the methods of multiple objects. This is to be seen in contrast with the need for multiple modalities for cooperative work quoted in the introduction of this paper and the flexibility requirements imposed by business process reengineering [DDJ+97].

3. Why distributed persistent agents are not enough

Despite significant differences in detail, models for **distributed agents** view a cooperative system as being composed of largely autonomous agents, each with a well-defined *responsibility*, *independent activity*, *private knowledge*, *memory* and *capabilities*. Moreover, agents are regarded as a unit of persistence and mobility.

Agents provide a system structuring concept appropriate for the decomposition of cooperative systems into smaller subsystems responsible for well-defined short-term or long-term tasks (claims processing, order management, shopping, information retrieval) which can be carried out either by a human or a software agent (demon, robot, script invocation, etc.) at a single site or involving a migration from site to site. As a concrete example for agent programming, the following Tycoon code fragment creates an autonomous `collectAgent`

```
let collectAgent(self :thread.T(Ok)) :Ok = begin
    let csDeptDB = agent.migrate(computingScience)
    let addressesOfProfessors =
       select p.address from p in csDeptDB.persons where p?professor
    let admin = agent.migrate(administration)
    admin.insertAll(addressesOfProfessors)
end
thread.fork(collectAgent)
```

Inside the body of the function, the parameter `self` is a handle for the thread that executes the function (unused in this example). Thread migration is accomplished by calls to the function `migrate` exported from the module `agent`. It (atomically) copies the current thread to the site designated by its argument, kills the currently executing thread and resumes thread execution at the remote site, returning a binding to local resources at the remote site, as defined by the type declared for that site [MMS95b]:

```
computingScience :Site(Tuple persons :Persons ... end)
administration :Site(Tuple insertAll(:Addresses):Ok  ... end)
```

Abstracting from the (significant) differences in typing and naming, this programming style is very similar to the one used in Telescript [GM95a], Mole [Hoh95], Facile [Kna95] or Obliq [Card94].

Unfortunately, such a programming style does not scale well to cooperative information systems since it inherits many of the deficiencies described in the previous section: the agent accesses *directly* through methods (`admin.insertAll`) or even through unprotected database variables (`csDeptDB.persons`) the state at the destination site, instead of interacting on a peer-to-peer basis with another agent.

To overcome these deficiencies, we concentrate on the externally observable behavior of agents, namely their ability to sustain long-term, goal-directed *conversations* with other agents which is also an important mechanism to *coordinate* agents (synchronization, delegation, replication, ...). By forcing agents to interact through conversations only, some disadvantages of direct object bindings can be avoided:

- Conversations do not impair agent autonomy. By exchanging well-defined dialog content with copy semantics only, no private object bindings become available to the communication partner. Therefore, it is not necessary to introduce new binding mechanisms like the ill-defined concept of *object references* of Telescript which attempts to distinguish objects belonging to the client and the server, respectively.
- Conversations do not restrict agent mobility since local and remote agents are treated uniformly. Therefore, it is possible for an agent to migrate between address spaces while sustaining persistent conversations, e.g. with its human *owner*.
- The agent system already provides a well-defined *concurrent* execution model. Contrary to direct object interactions, the coordination of multiple agents and conversations is encapsulated in the agent system layer and there is no necessity for application-level synchronization in cases where shared resources are manipulated. The application programmer is thus shielded from much of the complexity that arises in highly concurrent agent systems
- Conversations are based on static process descriptions (conversation specifications) which are first-class run-time objects available to both communication partners as soon as a conversation is initiated. This makes it possible to detect mismatches between the client and server view early. (e.g., "claims can only be settled after a contract has been signed.").

4. The model of business conversations

The leitmotiv of the Business Conversation model are speech acts between customers and performers [Wino87,FGHW88,MWFF92]. For example, an enterprise or a business unit is viewed as an agent that is involved in a number of (long-term) business conversations with other agents like customers, suppliers or government agencies. Within each of these conversations, each agent has a fixed role (either customer or performer). For example, an insurance broker is a performer for its customers and at the same time a customer for several insurance agencies.

Each business conversation can be decomposed into an ordered sequence of speech acts which can be classified into four phases that occur in the following sequence (see also Figure 1).

- **Request Phase**: The customer states the (business) goal to be achieved during the conversation. ("I want to insure my car").
- **Negotiation Phase**: A sequence of negotiation speech acts may be necessary to align the specific customer needs and the available performer services. Only if both partners agree on the common goal, a commit of the performer is reached which can be understood as a promise about his future activity.
- **Performance Phase**: The performer reports on the progress of and/or the completion of the requested activity to the customer. ("Here is your insurance card")
- **Feedback Phase:** This phase gives the customer the opportunity to declare its satisfaction with the service provided and may comprise the obligation for payment.

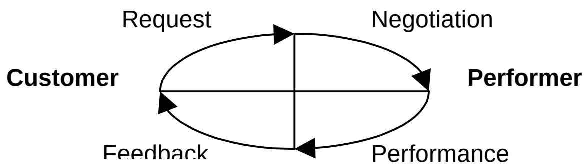

Figure 1: The four phases of a Business Conversation

In the Business Conversation framework, customers and performers can be either human or software agents which leads to an uniform treatment of four modes of agent/agent cooperation:

- **Application Linking**: Customer and performer are realized as two autonomous applications that synchronize via asynchronous message exchange.
- **GUI Management**: A human user interacts with a software system. Legal interaction patterns are described by the Business Conversation specification which are interpreted by a software system called *generic customer*.
- **Workflow Management**: A software system (as a customer) requests actions from a human user who is guided by a software system called *generic performer*.
- **Structured Message Handling**: The cooperative work of two human users can also profit from tool-supported message handling ensuring the adherence to pre-defined business rules.

The modality may change dynamically during an ongoing conversation. For example, similar to an automated telephone call center, standard requests could be handled by a software agent which transfers its conversation (i.e. a simple trace of past communication steps) to a human performer as soon as more complex or exceptional requests occur.

The speech acts (protocols) between customer and performer are constrained to be sequences of *structured dialogs* and they have to adhere to explicit *conversation specifications* which are essentially non-deterministic automata where each state is enriched with type information to describe the admissible contents and requests in this particular dialog step.

5. Implementation of business conversations

The system implementation consists of a polymorphically-typed framework written in the Tycoon persistent and distributed programming environment [MSS95], exploiting mobile persistent threads described in [MMS96b]. The framework components are replicated at each agent-enabled network site and are therefore viewed as an *ubiquitous* infrastructure.

A conversation specification is a contract between two agents since it constrains the behavior of the performer and provides a promise to the customer. A conversation specification can express type constraints (the structure of the documents sent and received) but also as state-dependent constraints on the conversation history (e.g., claims can only be settled after a contract has been signed and the first payment has been received). A more software-oriented example is the state-dependent specification that a customer will never execute a *pop* operation on an empty stack.

Technically speaking, conversation specifications are typed, persistent and mobile objects that are created bottom up using constructors of their respective classes, for example:

```
let contract = RecordContentSpec.new()
  .add("name"  AtomicContentSpec.new(String))
  .add("first"  AtomicContentSpec.new(String))
  .add("birthday"  AtomicContentSpec.new(Date))
  .add("method of payment" MultipleChoiceSpec.new().add( ... ) )
let contractDialog = DialogSpec.new(contract)
  .addPossibleRequest("Accept" confirmationDialog)
  .addPossibleRequest("Reject" negotiationSubConv)
  .addPossibleRequest("Reject" noAgreementDialog)
let carInsuranceConversationSpec = ConversationSpec.new("carInsurance")
  .add("Welcome" welcomeDialog)
  .add("Contract" contractDialog)
```

We are developing tools to generate a conversation specification from a textual or graphical representation of the dialog graph or to receive it from a remote agent, for example, a *conversation broker*.

A conversation specification (car insurance) is a dictionary of named dialog specifications with a distinguished initial dialog specification which describes the initial state for both communication partners. A dialog specification can be are either a concrete dialog specification (contract) or a subconversation specification (negotiation on contract details).

A concrete dialog specification consists of a record dialog content specification and a (possibly empty) set of request specifications available for this dialog. A record dialog content specification aggregates named content specifications (name, birthday, method of payment) which in turn (and recursively) can be either atomic (integer, string, ...), record, variant, sequence, single choice and multiple choice content specifications. In this way, content specifications define a simple monomorphic type system.

Once a Business Conversation specification object has been created, performer and customer agents which adhere to this specification can be defined by rules consisting of an event and a piece of code. This code typically triggers state transitions, initiates secondary conversations or performs actions through effectors attached to the agent. The code is parameterized by a conversation descriptor which holds conversation-specific data (identity of the conversation partner, contents and requests of all preceding dialog steps, etc.). This descriptor can be expanded by agent-specific data (invisible to other agents) and greatly simplifies the management of concurrent conversations with multiple customers and performers, respectively. On termination, the code attached to an event has to return an object that matches the constraints expressed in the corresponding conversation specification which is then transferred back to the customer.

An agent can support multiple customer and performer roles (e.g. performer for car insurance, performer for freight insurance). For each of these roles there exists a set of customer and performer rules, respectively.

A performer rule is defined for a particular request of a particular dialog specification (`contract.accept`, to be issued by a an agent in a customer role) and has to return an object of class dialog while a customer role is defined for a particular dialog specification (e.g., `contract`, to be generated by an agent in the performer role) and has to return an object of class request which has to be one of the requests admissible in this dialog step (e.g., `accept`, `reject`, `explain`).

An active conversation links exactly one customer role and one performer role of an agent. It also implements exactly one conversation specification and aggregates an ordered list of history elements which record the past dialog steps and requests of this conversation.

Customer and performer neither share data nor code or thread state. In particular, the conversation specification and the conversation trace are duplicated in the address spaces of both communication partners which ensures a high degree of agent autonomy and mobility.

We are currently working on a formalization of the refinement relationship between conversation specifications (similar to subtyping) and plan to investigate whether it is possible to build a static type checker that guarantees that a software agent generates (at run-time) only conversation traces which conform to a given static conversation specification.

References

[Card94] Cardelli, L. Obliq: *A Language with Distributed Scope*. Technical report, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, June 1994.

[DDJ+97] De Michelis, Giorgio, Dubois, Eric, Jarke, Matthias, Matthes, Florian, Mylopoulos, John, Papazoglou, Mike, Pohl, Klaus, Schmidt, Joachim, Woo, Carson, and Yu, Eric. *Cooperative Information Systems: A Manifesto*. In: Papazoglou, Mike P. and Schlageter, Gunther (Eds.). *Cooperative Information System: Trends and Directions*. Academic Press, 1997.

[FGHW88] Flores, F., Graves, M., Hartfield, B., und Winograd, T. *Computer Systems and the Design of Organizational Interaction*. ACM Transactions on Office Information Systems, 6(2), 1988, 153-172.

[GM95a] General Magic's Telescript home page. http://www.genmagic.com/Telescript/, 1997.

[Hoh95] Hohl, Fritz. *Konzeption eines einfachen Agentensystems und Implementation eines Prototyps*. Diplomarbeit, Universität Stuttgart, Abteilung Verteilte Systeme, August 1995.

[Kna95] Knabe, Frederick Colville. *Language Support for Mobile Agents*. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA 15213, October 1995.

[MaSc94] Matthes, F. and Schmidt, J.W. *Persistent Threads*. In: Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB}, Santiago, Chile, September 1994, 403-414.

[Math96] Mathiske, B. *Mobility in Persistent Object Systems*. Dissertation, Fachbereich Informatik, Universität Hamburg, Germany, May 1996.

[Matt93] Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993.

[Matt97a] Matthes, Florian. *Business Conversations: A High-Level System Model for Agent Coordination*. In: Proceedings of the Sixth International Workshop on Database Programming Languages, Estes Park, Colorado. Springer-Verlag, August 1997.

[MMS95b] Mathiske, B., Matthes, F., and Schmidt, J.W. *Scaling Database Languages to Higher-Order Distributed Programming*. Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy. Springer-Verlag, September 1995.

[MMS96b] Mathiske, B., Matthes, F., and Schmidt, J.W. *On Migrating Threads*. Journal of Intelligent Information Systems, 1996.

[MSS95] Matthes, F., Schröder, G., and Schmidt, J.W. *Tycoon: A Scalable and Interoperable Persistent System Environment*. In: Atkinson, M.P. (Ed.). Fully Integrated Data Environments, Springer-Verlag (to appear), 1997.

[MWFF92] Medina-Mora, R., Winograd, T., Flores, R., and Flores, F. *The Action Workflow Approach to Workflow Management Technology*. In: Turner, J. und Kraut, R. Proceedings of the Fourth Conference on Computer-Supported Cooperative Work. ACM Press, 1992, 281-288.

[OHE96] Orfali, Robert, Harkey, Dan, and Edwards, Jeri. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1996.

[Wino87] Winograd, T.A. *A Language/Action Perspective on the Design of Cooperative Work*. Technical Report No. STAN-CS-87-1158, Stanford University, May 1987.

Author:

Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg
21071 Hamburg
Tel.: 040 / 7718 3460, Fax: 040 / 7718 2515, E-Mail: f.matthes@tu-harburg.de