A Model-Centered Electronic Commerce Middleware

Giovanni Vigna[1,2] and Leonardo Bonomi[2]

[1] Politecnico di Milano
P.za Leonardo da Vinci 32
20133 Milano, Italy
vigna@elet.polimi.it
[2] CEFRIEL
Via Fucini 2
20133 Milano, Italy
bonomi@cefriel.it

Abstract. As electronic business transactions over the Internet become more popular, users and markets pose new requirements on both electronic commerce services and the applications implementing them. These requirements include support for multiuser business protocols, cooperation among users, and real-time (multimedia) interaction. The development of this kind of electronic commerce applications is a complex, costly, and time-consuming activity. Electronic commerce middlewares are an approach to support the development of these applications by providing a set of commerce-oriented functionalities that can be integrated into an application. Nonetheless, these platforms provide little support or guidance during the *conceptual design* of the application. This paper presents *Jecom*, a new approach to the development of distributed, cooperative electronic commerce applications. *Jecom* defines an application reference model, based on roles and state transition diagrams; furthermore, the model is directly supported by a distributed middleware. This way, *Jecom* supports *both* the conceptual design and the implementation of complex electronic commerce applications. This integrated support reduces the time and costs of application development.

1 Introduction

Electronic commerce has become a reality and real money is presently being invested to develop electronic commerce applications and infrastructures. The increasing interest of users and markets in electronic commerce is giving a new impetus to research on advanced electronic commerce services. These services pose new requirements on the applications that implement them. In fact, services tend to differentiate and become more sophisticated, overcoming the traditional *seller/purchaser* business model: advanced services may involve multiple users playing different roles and may require long-lasting transactions. Thus, applications must be structured to manage complex real-time interactions with a number of users and processes distributed over a global network.

Developing advanced electronic commerce applications that satisfy these requirements is a challenging task. The application developer must design complex business protocols, implemented by distributed architectures. Therefore, the design and actual implementation of the application can be complex, tricky, and, as a consequence, error prone.

Presently, the development of these applications is supported by commerce-oriented middlewares. Electronic commerce middlewares are platforms that provide domain-specific functionalities and infrastructures that can be exploited to build electronic commerce applications. For example, a middleware may provide an object-oriented framework that defines some useful abstractions (like wallets and electronic cash registers), an information infrastructure (like a directory that stores user and product information), or some libraries (like cryptographic functions that can be used to implement cryptographic protocols).

Even though existing middleware platforms reduce the implementation effort, they do not support a well-defined application conceptual model. Therefore, they provide little guidance during the application conceptual design and they are not effective in providing application-specific development support (like protocol analysis or automatic code generation).

This paper presents *Jecom* (Java-based Electronic Commerce Middleware), a model-centered middleware platform that supports an integrated approach to the development of advanced electronic commerce applications. *Jecom* overcomes some of the limits of the existing middlewares by providing:

1. a well-defined conceptual model of the application;
2. a middleware that supports the model directly.

The proposed application model provides reference abstractions that guide the application designer, simplifying the conceptual design of the application. The model-centered middleware supports application implementation in a focused and effective manner.

This paper is structured as follows. Section 2 presents some related work and introduces our approach. Section 3 describes the application model. Section 4 presents the architecture of the middleware. Section 5 presents a case study. Section 6 draws some conclusions and outlines future work.

2 Related work

Existing platforms that support the development of electronic commerce applications can be roughly distinguished into two families: *customwares* and *middlewares*.

Customware platforms provide the developer with a "bare" electronic commerce application that must then be customized using product-specific information. The way purchasers interact with sellers is predefined, as well as the metaphors (*shopping charts*, *malls*, etc.) used to guide the users towards the final purchase. These systems allow the merchant to create an Internet point of sale in a time- and cost-effective way. In fact, the development effort is usually limited

to the structuring of product data into catalogs and to the design of parts of the graphical interface, such as banners and logos. Examples of this type of environments are Microsoft's Merchant Server [7], IBM's Net.Commerce [8] and Open Market's Transact [9]. Even though these systems are suitable for simple applications, they fail in supporting the development of electronic commerce applications involving several parties and characterized by complex business models.

Middleware platforms aim at overcoming the customware approach, supporting the development of complex electronic commerce applications. Middleware platforms usually come in the form of some run-time support and a set of APIs that allow the application to access the environment services. Services can be composed and integrated in a flexible way, allowing the implementation of arbitrarily complex applications. On the other hand, since there is almost no supported (or enforced) application model, the developer must take care of all the design and structuring aspects of the application development. The characterization of the different parties (users and processes) involved, the way their interaction is managed, and the selection and composition of the available services are left in the hands of the developer. Examples of these systems are CommerceNet's Eco System [10], and Sun Microsystems's Java Electronic Commerce Framework [1]. The former is a cross-industry project aimed at building a *framework of frameworks* to achieve interoperability among platforms belonging to different vendors. The latter is an open platform for supporting secure electronic commerce transactions on the Internet. The JECF provides a set of libraries for purchasing, banking, and finance, and implements facilities like wallets, and cryptographic engines. Both platforms are based on Java [2] and CORBA IIOP.

The middleware approach is also followed by the research projects OSM [5] and SEMPER [11]. Both are part of the European Community ACTS Program and aim at addressing the issues related to an electronic marketplace. OSM has been developed on the basis of a previous project, called COSM [6], and focuses on the management of *contracts* and *commercial services*, providing facilities for cataloging and brokering. SEMPER proposes an architecture that addresses the security requirement of an advanced business environment.

Jecom takes a different perspective: instead of focusing on the design and management of electronic marketplaces, it aims at addressing the problem of business application development. To achieve this goal, *Jecom* extends the middleware approach in two ways. First, it defines a sound application model, oriented towards multi-user complex business protocols. The model provides a reference during the conceptual design of the application. Second, it realizes a middleware platform centered on the model. This means that the middleware services are designed to provide support for the abstractions defined by the model and the model itself is used as the basis to provide automated support to the application's development process.

By providing a model-centered middleware, *Jecom* allows the developer to deliver complex applications in a timely and cost-effective manner as in the customware approach, while maintaining the flexibility and openness that characterize the middleware approach.

The application model and the middleware architecture will be presented in the next two sections.

3 A distributed application model

In *Jecom* an application is an entity whose task is to coordinate a set of users to carry out a business transaction. Users participating in an application play different *roles*. A role identifies a set of capabilities and responsibilities. For example, in an application that implements a simple purchase transaction, we can identify at least the role of the *purchaser* and the role of the *seller*. The former has to select the object to be purchased and will eventually pay for the purchase. The latter has to provide information about the object (e.g., its cost), and to produce a receipt for the purchase.

The main task of an application is the coordination of the different roles. As a consequence, its architecture is composed of a central component, called the *coordinator*, and a set of components, called the *interfaces*, one for each role involved in the application (see Figure 1).

The *coordinator* is the component responsible for the enactment of the application's business protocol and for the coordination of the parties involved. It receives *event notifications* from both the *Jecom* environment and the interface components, and performs *actions* basing on the type of event, the source of the event, and its current state. The coordinator's state evolves as a consequence of the actions executed.

From the above characterization follows that the coordinator behavior can be expressed in a intuitive way by using *state transition diagrams*. Each diagram state represents a state of the coordinator. Transitions are labeled with pairs $\langle source, event \rangle$ where *source* can be one of the application's roles or the predefined *system* source, and *event* is the name of an event. Each transition has an associated *action* that may produce side-effects or send events to the system and/or to the user interfaces. Events that are sent as a consequence of action execution are listed as annotations to transition labels. Each annotation is a pair $\langle dest, event \rangle$ where *dest* denotes the event destination and *event* is the name of the produced event.

During execution, when the coordinator receives an event e from source s, if there is a transition labeled with $\langle s, e \rangle$ starting from the current state, the action associated to the transition is executed. This may possibly trigger a state change and/or event dispatching.

Interfaces are the means of interaction between the human actors and the application. Interfaces are a source of and a destination for events. They interact with users and produce events for the coordinator and, in turn, receive events from the coordinator and interact with the human user to satisfy requests, to communicate messages, or to notify the user of changes in the application's state. As for the coordinator, interface components can be described in terms of state transition diagrams. Each state of the diagram represents a state of the interface. Transitions are labeled with source/event pairs, and have an associated action. Events may come from the user (e.g., a request to submit a form) or from the coordinator (like the request to display a message). When an event is received, if,

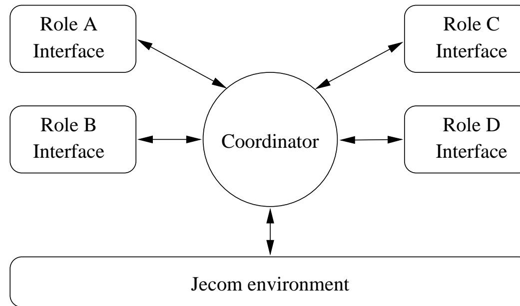

Fig. 1. Application abstract architecture.

starting from the current state, there is a transition labeled with the corresponding source/event pair, the action is executed. The action may produce a state change or may send events to the user and/or the coordinator.

The introduction of an application model based on state transition diagrams provides some relevant advantages. First of all, the model defines a framework for the conceptual design of the application and the corresponding business protocol [3]. The design activity is guided by the concepts of roles and coordination among roles. While the components' behavior can be totally arbitrary, the application constituent abstractions are predefined. Thus the model provides guidance while maintaining flexibility and openness. Second, since state transition diagrams are a *formal* notation with a precise semantics, the application description can be used as the basis for protocol analysis [4] and automated support to the application development process. For example, the *Jecom* environment is able to generate automatically the overall structure of the application code starting from the description of its components. Hence, the developer has just to design and implement the data structures that maintain the application's state and the actions that modify it. In addition, state transition diagrams are an abstract, compact way to describe the application behavior. Therefore they provide an effective documentation of the application design.

4 An electronic commerce middleware

Jecom provides the typical functionalities of electronic commerce middlewares (payment, notary, security, etc.) while it extends the traditional approach in two directions:

1. it implements a *model-centered* architecture, based on the approach described in Section 2;
2. it provides a set of primitives that support the cooperation and the real-time interaction among the users involved in a business transaction.

Jecom maps the model described in Section 3 on the concepts of *application class* and *application instance*. An application class is the static representation

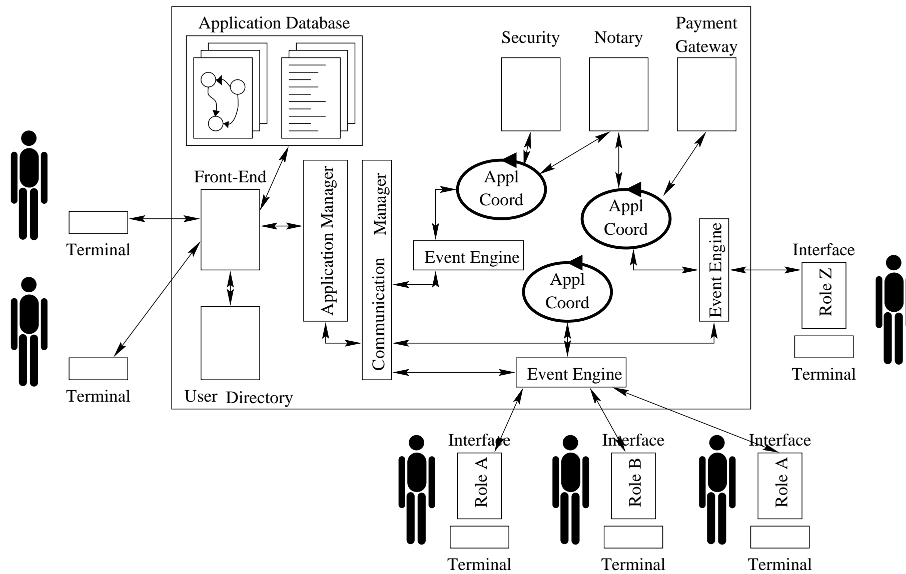

Fig. 2. Middleware architecture.

of an application. An application class is composed of the code for the coordinator and for each interface component, together with the specification of the application in terms of roles involved and state transition diagrams describing the application behavior. Code modules and state transition diagrams are stored by the *Application Database*. Application instances are the run-time representation of applications. They are processes that execute an application class code and enact a particular business transaction. For example, while an application class may implement a generic *assisted purchase* process, an application instance may represent a transaction involving user A acting as the purchaser of a computer laptop, user B acting as a computer hardware expert that provides advice to A, and user C as the merchant.

As prescribed by the application model, application instances are composed of a *coordinator* and of an *interface* for each different role. The coordinator component is a process running within the *Jecom* platform (see Figure 2). It interacts with both the middleware and the interfaces. Interface components are software modules dynamically downloaded on request in the user's remote terminal. In the current implementation, terminals are Web browsers and interfaces are Java applets[1].

A *Jecom* user accesses middleware services and applications through its terminal. Initial interaction with the middleware is mediated by the *Front-End* component. After the user has authenticated himself/herself, he/she browses the cur-

[1] To assure an adequate software portability, every middleware component, including the code corresponding to the application classes, has been implemented in *pure Java* [2].

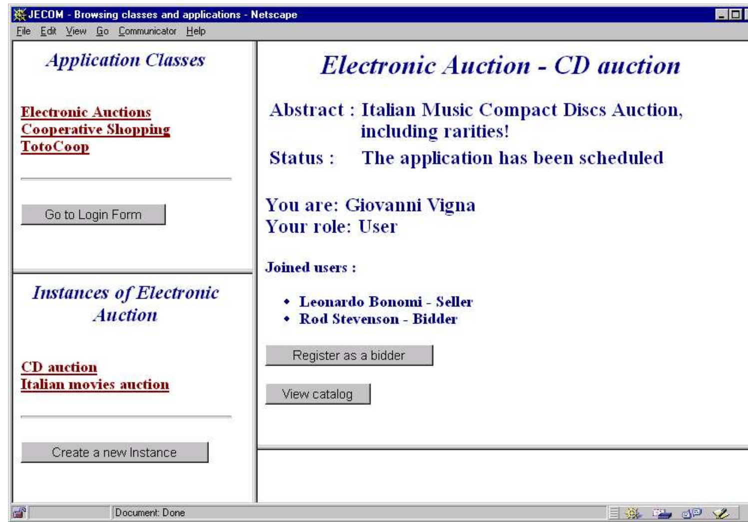

Fig. 3. Web-based interface.

rently registered application classes. Once an application class has been selected, the corresponding application instances are shown to the user. Then, he/she can: (i) create a new instance of an application class; (ii) join an existing application instance with a particular role. Both operations can be performed only if the user profile allows it. User profiles and other user-related information are stored in the *User Directory*.

The interface presented to the user is dynamically built by the Front-End on the basis of the system status, the selected application class, the status of the selected application instance, and the role assumed by the user. Figure 3 shows a sample snapshot of the interface during the application selection process. The upper left corner shows the existing application classes. The user has already selected the *Electronic Auction* application class, causing the instances of the Electronic Auction application class to appear in the bottom left corner. The user has then selected the *CD auction*. As a consequence, the top right corner of the interface presents the current state of the instance (namely, *scheduled*) and the available operations. Since the user currently plays the *user* default role, the only available operations are *register* and *view catalog*.

Once the user has joined an application instance with a specific role, he/she can request the execution of the *operations* allowed to his/her role in the current application status as described by the corresponding state transition diagram. Operations are carried out by the *Application Manager* whose tasks include also starting applications and managing their interactions with the middleware platform. Even though a user may operate on an application instance through the *Front-End* and the *Application Manager*, after authentication, instance selection, and registration, further interaction is usually carried out through the specific interface associated with the role selected by the user during the subscription

process. The user can download the interface associated with a particular role using the *download interface* predefined operation. When requested, this operation triggers the retrieval of the interface code from the *Application Database* and its downloading into the user's terminal. At startup, the interface module connects directly to the application coordinator running on the platform and starts interacting.

Interaction among application instance modules (coordinator and interfaces) is event-based, as well as the interaction between the coordinator and the underlying middleware. Each application instance has an associated *event engine* that acts as a software bus. The application modules and the middleware itself may subscribe to the event engine for specific events. When the event engine receives an event from an application module or from the middleware, it multicasts the event to all the modules that subscribed for that event. For example, the middleware subscribes automatically for events notifying state changes: this way it keeps track of the current application status. Events exchanged among the application modules are usually managed in an automatic way. Event subscriptions are derived from the state transition diagrams describing the application components, and are updated as the modules state changes. In addition, the event infrastructure can be used explicitly by application developers in order to implement other types of interaction, such as message exchange or queries. Each application's event engine is connected to a platform-wide event bus, the *Communication Manager*. This module allows for inter-application and platform-wide event communication. In principle, this hierarchical structure could be extended to integrate several platforms.

By assigning the task of event/state management to the middleware run-time support, *Jecom* allows the developer to focus only on the implementation of the data structures that maintain the state of the application modules and of the elementary actions executed in response to events, letting all the flow control and the event management to the middleware, which can automatically carry out these tasks just on the basis of the formal description of the application.

5 Case study

In this section we present a case study that describes the development of an application following the *Jecom* approach. For the sake of simplicity, we do not detail every aspect of the application and the associated developing process. Rather, we will focus on those key features that represent the distinguishing characteristics of the proposed approach.

The case under exam is an *electronic auction*. The application can be informally described as follows. A seller organizes the auction event. He defines the items that will be offered. For example, he may decide to organize an auction of compact discs. Therefore he provides a list of the CDs that will be offered together with some presentation information. Users may register to the auction. When the seller starts the auction, an auctioneer notifies the users of the item being auctioned and its initial price. Then, users compete by proposing higher prices. If no bids are made, after a specified timeout the item is discarded. Otherwise, after

each bid a timeout is started. If no bids are received during the timeout, the item is assigned to the user that performed the last bid.

From this description we can identify three roles:

- The *seller*, who offers goods. For simplicity, this role is assigned to the user who creates the auction application instance. In a more elaborated model, one may decide to model a separated *manager* role whose task is just to organize the auction event.
- The *bidders*, who participate in the auction, bidding for any item they are interested in.
- The *auctioneer*, whose job is to notify bidders of the item auctioned, and to assign the item to the last bidder, after a timeout. This role can be carried out in a completely automated way. Thus, it will be played by a logical thread inside the application.

In addition, we use the *user* role that is a predefined *Jecom* role.

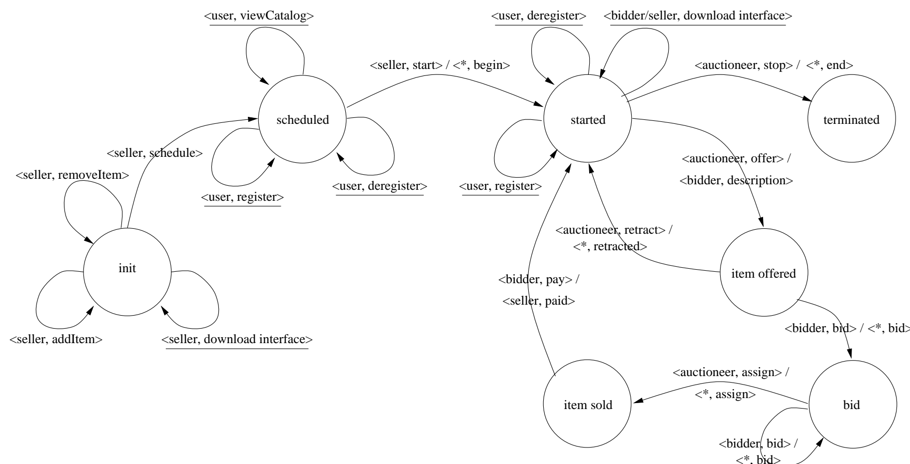

Fig. 4. Electronic auction

The application coordinator can be described by the state transition diagram shown in Figure 4. When a user creates a new instance of the electronic auction application, he/she assumes automatically the role of the seller. At startup, the coordinator is in state *init*. In this state there are a limited number of operations allowed: the seller can just add or remove items from the catalog associated with the auction. After the catalog has been prepared, the seller may schedule the auction for a particular date. Scheduling causes a transition from state *init* to state *scheduled*. In state *scheduled* users can view the auction catalog or register/deregister as bidders. Users that register themselves with the application before its start, will be notified at application startup. The auction starts when the seller performs the corresponding operation, namely *start*. The coordinator

switches to state *started* and broadcasts the *begin* message to all roles. Starting from state *started* the auctioneer selects the items proposed by the seller one by one. After each selection, the coordinator sends the description of the selected item to all bidders; the application then switches to the *item offered* state. In this state two events are acceptable: (i) the auctioneer retracts the item (meaning that no bidder has made an initial offer) and the coordinator status switches back to *started*; (ii) a bidder makes a first bid. In this case all the participants are notified and the state moves to *bid*. In state *bid*, bidders compete by offering higher prices. The auctioneer automatically assigns the item, if, after a timeout, there have been no higher bids. As a consequence, the coordinator state becomes *item sold*. In this state the coordinator just waits for the item to be paid. When the bidder pays, the seller is notified of the transaction, and the application state becomes *start*.

Note that even if the auctioneer role is played by the application itself, yet it is useful to represent the role in the application description. First of all because the auctioneer role is an important part of the auction protocol. If automatic analysis of the protocol must be performed, all roles and events must be explicited. Second, it is correct from an architectural viewpoint to represent the auctioneer as a separate entity (a different thread of execution). This supports separation of concerns and modularization.

Once the coordinator behavior has been described, the *Jecom* environment generates the code skeleton for the application coordinator. This code skeleton must be filled with the definition of the data structures that actually represent the application state (e.g., a table containing all the connected users), and the code implementing the actions (e.g., the code that updates the table when a bidder deregisters from the application). The events that are underlined are supposed to be managed by the middleware itself. Therefore users may perform these operations by using the *Front-End* and the Web-based interface (for example, Figure 3 shows the interface used to register with the auction).

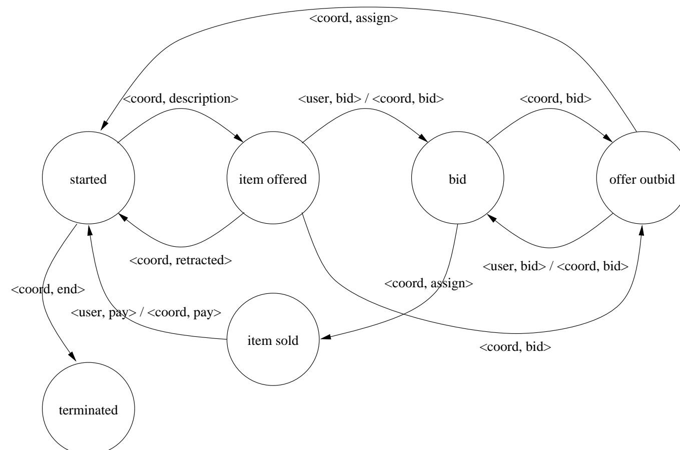

Fig. 5. Description of the bidder interface.

The same development process described for the coordinator is followed also for the interfaces. They are described with a state transition diagram that models the interaction of the module with the user (if any) and the coordinator. The state transition diagram for the bidder interface is represented in Figure 5, while a snapshot of the corresponding applet interface[2] is presented in Figure 6.

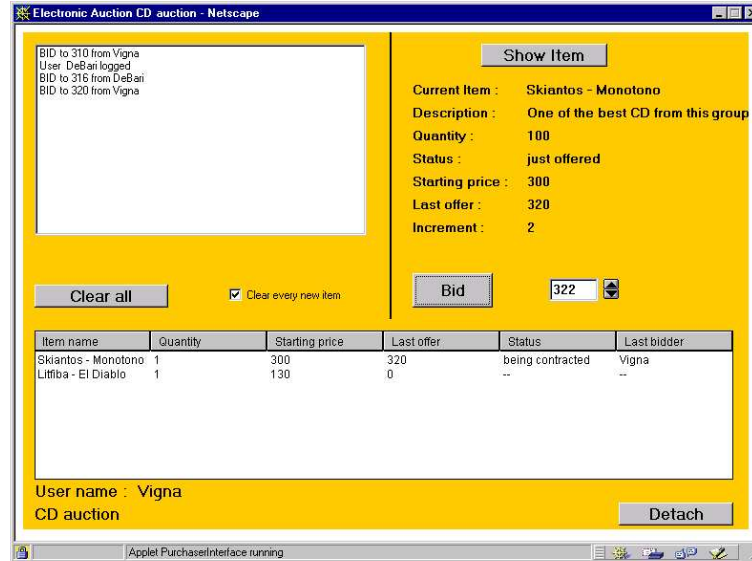

Fig. 6. Bidder interface applet.

6 Conclusions and future work

The evolution of electronic commerce services poses new requirements on applications. These applications must implement complex business protocols involving several users playing different roles. These users have to cooperate and interact in order to carry out a business transaction.

Presently, the development of complex electronic commerce applications is supported by middlewares. These platforms provide a set of useful predefined functionalities but they do not support a well-defined application model.

This paper has presented *Jecom*, a new integrated approach to the development of complex, cooperative applications. *Jecom* provides both an application model based on events and state transition diagrams and a middleware centered

[2] The middleware will not take care of producing code for the graphic interface. The middleware generates the skeleton for the state/transition management. The mapping of graphic input/output — as buttons pressed or messages printed in text areas – into input/output events is left to the programmer.

on this model. This integrated approach provides the developer with guidance from the conceptual design to the final implementation of the application, reducing the time and costs associated to application development. The system has been used to develop a number of test applications for a major national Telecom provider, including two different types of electronic auction, a cooperative web shopping application, and a cooperative soccer gambling application.

Our experience in using the *Jecom* platform showed that developers are provided with effective support. Developers can tackle design issues within a clear framework, that simplifies design and review tasks. In addition, implementation times are actually reduced because part of the code is generated by the platform and some consistency checks on the business protocol are performed automatically. Our experience showed also that the use of an explicit representation of the application behavior allowed the developers to implement variations of the same kind of application in a very short time with reduced effort.

Jecom is a functioning prototype. Current research efforts are focused on system reengineering, and in building graphic support for application design.

As future work, we plan to introduce support for business protocol design and verification. A first very simple prototype that perform consistency checks on the interaction among the coordinator and the interfaces is under development.

References

1. A. Coleman. Java Commerce: A Business Perspective. Sun Microsystem's White Paper, 1997.
2. J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, October 1995.
3. Taligent Inc. Building Object-Oriented Frameworks. Taligent White Paper, 1994.
4. R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, (7):pp. 79–130, 1994.
5. S. McConnell, M. Merz, L. Maesano, and M. Witthaut. An Open Architecture for Electronic Commerce. OSM Technical Report, February 1997.
6. M. Merz, B. Liberman, K. Müller-Jones, and W. Lamersdorf. Interorganizational Workflow Management with Mobile Agents in COSM. In *Proceedings of PAAM '96*, London, April 1996.
7. Microsoft Merchant Server. *http://www.microsoft.com/merchant/*, 1997.
8. IBM Net.Commerce. *http://www.software.ibm.com/commerce/net.commerce/*, 1997.
9. Open Market Transact. *http://www.openmarket.com/transact/*, 1997.
10. J.M. Tenenbaum, T.S. Chowdhry, and K. Hughes. Eco system: An internet commerce architecture. *IEEE Computer*, May 1997.
11. M. Waidner. Development of a Secure Electronic Marketplace for Europe. In *Proceedings of ESORICS '96 (4th European Symposium on Research in Computer Security)*, number 1146 in LNCS. Springer-Verlag, 1996.