Pergamon

Information Systems Vol. 25, No. 2, pp. 135–156, 2000

Printed in Great Britain
0306-4379/00 $20.00 + 0.00

A PROCESS-ORIENTED AND CONTENT-BASED PERSPECTIVE ON SOFTWARE COMPONENTS

HOLM WEGNER, PATRICK HUPE and FLORIAN MATTHES

Software Systems Group (STS), Hamburg University of Technology, D-21071 Hamburg, Germany

(Received 15 December 1999; in final revised form 22 March 2000)

Abstract — Commercial software component models are frequently based on *object-oriented* concepts and terminology (e.g. interfaces, classes, methods, messages, events) with appropriate binding, persistence and distribution support. In this paper, we argue that a *process-oriented* and *content-based* view on cooperating software components based on the concepts and terminology of a language/action perspective on cooperative work [41] (e.g. actors, roles, conversations, speech acts, conversation histories) with a tight coupling between process- and content-model provides a more suitable foundation for defining software components in business applications. This especially applies to the emerging fields of software for information brokers, value-adding services, digital libraries etc. We first explain the relationship between object-oriented and process-oriented component modeling, then describe our view on component definition and finally illustrate it using two recently executed industrial case studies. We also report on our experience gained in developing a class framework and a set of tools to assist in the systematic process- and content-oriented development of business application components. In particular, the paper addresses consistency checking of business components based on temporal properties and consistency-preserving composition of process fragments.

Key words: Process Models, Content Models, Software Components, Cooperative Information Systems

1. INTRODUCTION AND RATIONALE

Organizations utilize information systems as tools to support cooperative activities of employees within the enterprise. Classical examples are back-end information systems set up to support administrative processes in banks and insurances, or processes in enterprise resource planning.

Driven by various factors (availability of technology, group collaboration issues and organizational needs), there is a strong demand for more flexible and decentralized information system architectures which are able to support

- cooperation of humans over time (persistence, concurrency and recovery),
- cooperation of humans in space (distribution, mobility, online and offline users), but also
- cooperation of humans in multiple modalities (batch processing, transaction processing, asynchronous email communication, workflow-style task assignment, ad-hoc information sharing).

Another crucial observation is the fact that cooperation support can no longer be restricted to employees (intra-organizational workflows) but also has to encompass inter-organizational workflows involving customers, suppliers, tax authorities, banks, etc.

The objective of our research is to identify abstractions and architectural patterns which help to design, construct and maintain software components in such a "cooperative information system" environment [8, 9, 29].

Currently, there are two different views on what constitutes a good approach to modeling business processes: on the one hand the object-oriented approach originating from software engineering and on the other hand a process- and content-based approach that allows for designing business processes as first class objects. Our model integrates both approaches to construct information systems where the components are based on the modeling language UML [33] following the Unified Software Development Process [19] and the business processes are modelled on top of them.

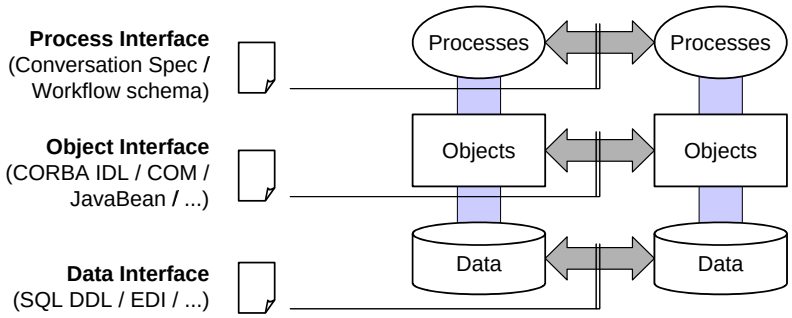

Fig. 1: Three Approaches to Component Definition in Business Information Systems

In Section 2, we argue that the successful step in information system engineering from data-oriented to object-oriented information system modeling should be followed by a second step from object-oriented to process-oriented system engineering. Only a process-oriented perspective allows software architects and organizations to identify and to reason about actors, goals, cooperations, commitments and customer-performer relationships which are crucial in a world of constant change to keep the organizational objectives and the objectives of the supporting information systems aligned.

In Section 3, we illustrate our approach to component definition using two case studies from projects with German software companies.

Details of the underlying process and system model are given in Section 4. We report on our experience gained in developing a class framework and a set of tools to assist in the systematic process- and content-oriented development of business applications. This part of the paper also underlines that our perspective fits well with today's object-oriented language and system models.

Section 5 describes the use of conversation specifications to support consistency checking of business rules based on temporal properties. The paper ends with the description how to compose business applications from process fragments in a flexible way.

2. APPROACHES TO COMPONENT DEFINITION IN INFORMATION SYSTEMS

In this section, we briefly review the evolution of information system architectures to highlight the benefits of a process-oriented approach to component definition.

Figure 1 summarizes the main issues of this section:

- Interaction between system components in an information system can be understood at three levels, namely at the level of *data* access, *object* interaction and at the level of *process* coupling.

- At each level, interfaces between system components should be declared in an abstract, system-independent syntax which also provides a basis for the systematic implementation of vendor-independent middleware.

- Abstract concepts of the interface language are mapped to concrete implementation concepts of the participating system components.

- A higher-level interface language includes concepts of the lower-level interface language but often imposes additional restrictions on the use of these concepts. For example, CORBA IDL provides data attributes and attribute types similar to record attributes and SQL domains, but also provides mechanisms for data encapsulation at the object level.

2.1. Data-Oriented Component Definition

Database research and development traditionally has focused on the entities and relationships of an information system and led to the development of conceptual, logical and physical data models, generic systems and tools as well as software development processes to support the analysis, design and efficient implementation of (distributed) data components.

Today, virtually all organizations have conceptual models of the information available in their back-end information systems and systematic mechanisms to develop and change applications to satisfy new information needs.

For information stored in relational databases, SQL as the "intergalactic dataspeak" [39] provides both, a language for the exchange of vendor-independent data description (schemata, tables, views etc.) as well as a language for (remote) data access (via APIs like ODBC, JDBC etc.), see also Figure 1. Moreover, SQL databases provide generic tools for interactive and programatic data access, authentication and access-control which are crucial for the safe re-use of data components in different parts of the enterprise.

2.2. Object-Oriented Component Definition

In modern client-server architectures and in cooperating information systems, the information assets of an enterprise are modeled (and often implemented) as collections of distributed and persistent objects interacting via messages and events.

The interfaces for these components are defined as enriched signatures in object-oriented languages using concepts like interfaces, classes, objects, (remote) references, attributes, methods, qualifiers, subclasses, events, exceptions etc. These object and component models (CORBA, DCOM, JavaBeans, etc.) describe the interaction between components by (a)synchronous method invocation extending and adapting the semantics of dynamic method dispatching in object-oriented programming languages (see also Figure 1). Moreover, an object-oriented component interface definition constitutes a contract between the user of this component and the implementor of the component. They both have to agree on the set of valid messages and on the argument types of these messages.

The advantage of object-oriented component models over pure data models on the one hand and over purely procedural (remote) function libraries on the other hand is their ability to describe semantic entities which are already meaningful at the analysis and the design level. The often quoted classes `Customer` and `Shipment` are examples for such high-level entities.

2.3. Process-Oriented Component Definition

An object-oriented component definition is "richer" than a data-oriented component definition since the methods (`shipment.create`, `shipment.addItem`, etc.) provide a more suitable protocol for the interaction between a client and a server component than a sequence of raw SQL statements working on a shipment table.

However, we still see the following deficiencies of object-oriented component definitions which call for an even richer process-oriented component definition:

- The interface of a software component rarely consists of a single object interface but of a large number of highly interrelated object interfaces[†]. Often it becomes necessary to restrict a specific client's access to only a subset of these classes and their methods.

- Frequently, it is necessary for a server to manage multiple execution contexts, for example, one for each concurrent client session. Clients then often have to pass a *session* handle as an extra argument to each of these methods. This session handle is also required for transactional database access.

[†]For example, Microsoft Word is a software component which has several hundred interface classes.

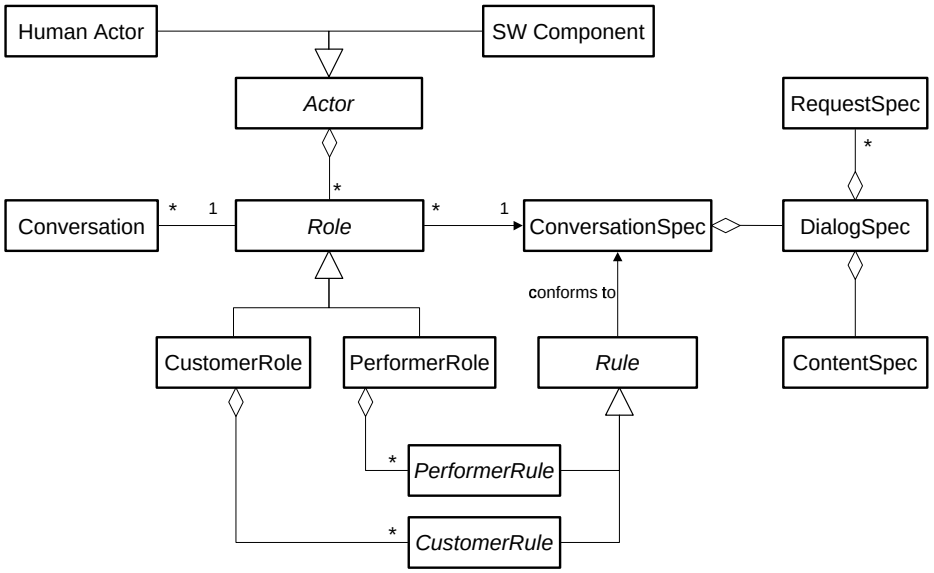

Fig. 2: Model Concepts

- Especially in business applications, it is desirable to enforce restrictions on the admissible execution order of certain method calls ("A shipment can only be send after a payment has been received"). This order is typically defined relatively to a specific client session. Since component definitions consist only of interfaces, there is no way to express such semantic or process-related issues.

- The lifetime of such execution contexts tends to be longer than the lifetime of a server process. Therefore, it becomes necessary to make execution contexts first-class persistent and possibly mobile objects.

- If a large system is broken down into autonomous *concurrent* subsystems (a collection of *agents*), synchronization issues can arise during method invocation.

- Following the Unified Software Development Process, a business process is split into several use-cases which are not related to each other furthermore.

As a solution to these problems, we propose not to use object-oriented component interface definitions but *process-oriented* component interface definitions (which are based on a high-level content model) between subsystems of a business information system following the language/action perspective on cooperative work [41, 14, 36, 6]. In the rest of this section, we will explain our model [26, 28, 31, 20, 40, 29]. The model concepts are illustrated in Figure 2.

In a first step, we identify *actors* in a business information system. An actor can either be a human or a software component which may be an active process (thread, task, job etc.) in an information system. For example, a customer, an SAP R/3 application server and a Lotus Notes Domino Server can all be viewed as actors. This first step can be sufficiently supported by UML use-cases; UML actors are identical to actors in our model.

In a second step, we identify *conversation specifications* to describe long-term, goal-directed interactions between actors. For example, if a customer can browse through an internet product catalogue on a Lotus Notes Domino Server to place an order online, we identify a conversation specification called *Order*. We also assign *roles* to actors within a conversation. The actor that initiates the conversation (the online shopper) is called the *customer* of the conversation. The actor that accepts conversation requests is called the *performer* of the conversation (Lotus Notes).

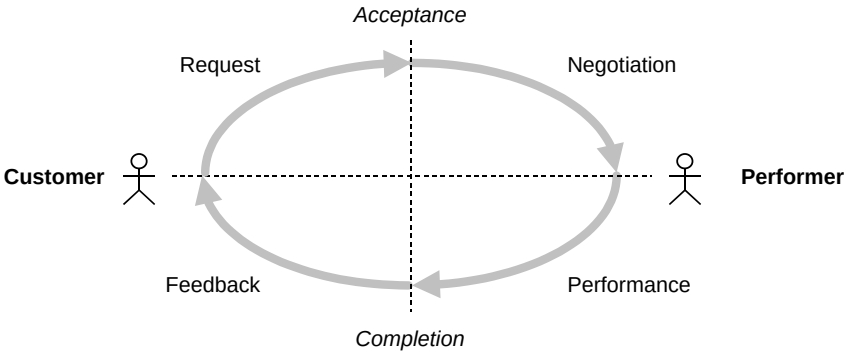

Fig. 3: Phases of Typical Customer-Oriented Conversations

A conversation is constituted of possibly several UML use-cases of one actor. The advantage compared to single use-cases is that the process description remains intact.

An actor can participate in multiple (possibly concurrent) conversations taking different roles. For example, Lotus Notes could use the services of SAP R/3 to check the availability of products (online or offline). Thus, Lotus Notes would be the customer of SAP R/3 for this particular conversation specification.

Next, we identify *dialog specifications* which are process steps within each of the conversations (login dialog, catalog view, item view, shopping cart dialog, credit card dialog etc. for online shopping). A dialog consists of a hierarchically structured *content specification* – the *content model* – plus a set of *request specifications* valid in this particular process step. For example, the shopping cart dialog could aggregate a set of shopping cart items (each of which consists of a part identification, a part name, a number of items and a price per item) plus a total, the name of the customer, VAT, etc. In the shopping cart dialog, only a restricted set of requests can be issued by the customer (leave shop, select payment mode, remove item from cart, etc.).

For each request specification in a dialog, there is a specification of the set of admissible follow-up (next) dialogs. If this set contains more than one dialog, the performer can continue the conversation at run-time choosing any of these dialogs. For example, the `addItemToShoppingCart` request in the item view dialog could either lead to the shopping cart view or to an out of stock error dialog.

It should be emphasized that a dialog specification fully abstracts from the details and modalities of dialog processing at run-time. For example, the dialog could be carried out synchronously via a GUI interface (Windows / Lotus Notes) or via HTTP or asynchronously via email or a workflow-style task manager.

Contrary to object interactions via message passing, this "form-based" or "document-based" style of interaction at the level of dialogs also fits well the (semi-)formal interaction between humans. For example, we are all used to a form-based interaction with public authorities. Another important observation is that this document based interaction also fits well with the emerging new technologies needed for digital libraries, information brokers and value-adding services in the internet, where less complex object models, but rich and expressive document models are needed.

We consider the ability to abstract from the modalities of an interaction (local / remote, synchronous / asynchronous, short-term / persistent, involving systems / involving humans) a major contribution of our process model since it enables us to uniformly describe a wide range of interactions between actors that are relevant for the enterprise.

Figure 3 illustrates the basic structure of a typical customer-oriented conversation. In the first step, the *request*-phase, a customer asks for a specific service of a performer (e.g. "I want to order a hotel-room"). In the second phase, the *negotiation*-phase, customer and performer negotiate their common goal (e.g. conditions of satisfaction, quality of service) and possibly reach an agreement.

To do this, several dialog iterations may be necessary. In the third phase, the *performance*-phase, the performer fulfills the requested activity and reports completion back to the customer ("we accepted your order"). The optional fourth phase is called *feedback*-phase and gives the customer a chance to declare his/her satisfaction, it may also contain the payment for the service.

This basic structure can already be found in data-oriented systems where a conversation roughly corresponds to a transaction that needs to be initialized, committed or possibly rolled-back.

It should be noted that we deliberately restrict our model to *binary* customer / performer relationships and that we do not follow established workflow models (from CSCW) that focus on process chains involving multiple performers from different parts of the enterprise. For example, in our model a workflow with three specialized performers could be split by a coordinator that takes a customer request and that initiates three separate (possibly concurrent) conversations with the three performers involved. Alternatively, one could construct a chain where each performer initiates a follow-up conversation after successful termination of his own process step.

This example also illustrates that conversation specifications are an excellent starting point for component definitions since they help to identify data and control dependencies. Moreover, it becomes much simpler to assign (data, behavior and also process) responsibilities to actors than it is the case for pure data- or object-oriented models.

This approach also narrows the gap between requirements analysis, done by UML-style use-cases etc., and architecture and design. Starting from the content-model, i.e. identified dialog specifications, designing an object model for involved business objects becomes simple. One could also see the dialog specifications as kind of conceptual object model in terms of UML.

To summarize, conversation specifications include content models (similar to complex object models) and behavior specifications (similar to methods in object models) and they provide *additional* mechanisms for process modeling:

- Actors and their roles in the network of conversations of an enterprise are modeled explicitly and at a high level of abstraction.
- The context of a request is modeled explicitly, too. For example, the history of a conversation is accessible.
- Requests can be restricted to certain steps (dialogs) within a process.
- It is possible to specify (aspects of) the dynamic behavior of the process through the set of follow-up dialogs defined for a request.

Finally, it should be noted that conversation, dialog, request and content specifications *can* be used as *static* interfaces between components. Only at run-time, conversations, dialogs, requests and contents are created *dynamically* as instances of these classes. This corresponds to the distinction between schema and database at the data level and the distinction between interface and object at the object level.

However, specifications can also be expressed as first-class objects in an implementation language [40]. This makes using and reusing of conversations much more flexible and permits generic, dynamic and reflective services [35], which is extremely valuable for the content model and thus for document-oriented services like digital libraries etc.

3. CASE STUDIES

In this section, we illustrate our approach to component definition using two case studies from projects with German software companies, namely a bug tracker system implemented with Microsoft ASP and MTS [5] and a hotel reservation system with an SAP R/3 backend using SAP BAPIs [34].

The goal of these projects was to investigate whether the abstract process component model described in [26, 28] and successfully implemented in persistent programming languages [20, 40] is also a suitable basis for the implementation of process components using commercially relevant technology.

Model Concept	Implementation Concept			
	SAP R/3 Dynpro Technology	**SAP R/3 BAPI Technology**	**Java Server Technology**	**Microsoft ASP Technology**
Agent (Component)	R/3 Application Server	R/3 Application Server	HTTP-Server with Servlets	HTTP-Server with ASP Extension
Performer Role	Collection of related Dynpros	Collection of related BAPIs	Collection of Servlets	Collection of ASPs
Customer Role	n.a.	n.a.	n.a.	n.a.
Conversation	R/3 Dynpro	Client Session / SAP Transaction	Servlet-Session	ASP-Session
Dialog	Dynpro Screen	n.a.	HTML-Form	HTML-Form
Request	Modification of GUI Variable	BAPI Method Invocation (RFC)	HTTP-Request	HTTP-Request
Content	Dynpro Screen Field	BAPI Method Arguments	HTML-Document	HTML-Document
Rule	PBO / PAI-Module	Implementation of BAPI (RFC)	Servlet	Embedded Script Code
Conversation Specification	EPC Description of Dynpro	n.a.	n.a.	n.a.
Request Spec	EPC Event	n.a.	HTML-Form	HTML-Form
Content Model	(I/O values of EPC transition)	n.a.	DTD	DTD

Table 1: Mapping of Process Component Model Concepts to Implementation Concepts

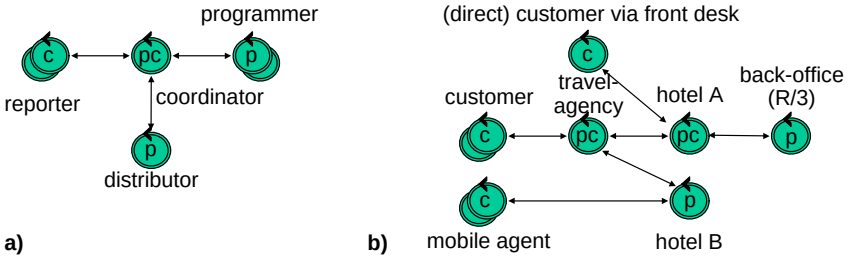

Fig. 4: Agents and Conversations of the Case Studies

The conceptual basis for these projects is summarized in Table 1 that shows the (rough) correspondence between the abstract model concepts and the implementation concepts of the respective languages or systems used to systematically realize the model concepts. For convenience, we also added a column describing the relationship to Java HTTP-Servlets. Several cells in the table are marked as "not applicable" (n.a.), since some of the systems lack the required modeling support. However, these concepts can be emulated by a systematic use of other language constructs[†].

Figure 4 summarizes the agents[‡] and conversation specifications of the case studies. In this diagram, an agent is indicated by a circle with an arrow. A conversation specification between two agents is indicated by a bidirectional arrow connecting the agent icons. If there are multiple agents that are based on the same conversation specifications, the icons of these agents are stacked. Each agent is annotated with the letter(s) P or C depending on his role(s) in the participating conversations (performer or customer role).

Figure 4 is described in more detail in the following subsections. At this point, we should simply note that some of the agent *patterns* depicted in this figure (e.g., coordinator, mediator, broker) tend to appear in multiple application domains.

[†]This is similar to the situation in CORBA where IDL interfaces and modules can also be mapped to simple imperative programming languages like C or COBOL that lack methods and dynamic binding.

[‡]The term *agent* is used as a synonym to *actor* in this section.

3.1. A Workflow Manager for Software Bug Tracking

The first example illustrates how conversation specifications can be used to structure the interaction between multiple human agents within an enterprise.

At STARDIVISION INC., a cooperative information system was created for tracking and removing bugs in software (see diagram (a) in Figure 4). There are three kinds of human actors and one software actor:

- A reporter is a human agent (a member of the support staff or a person answering the hotline) that is a customer in a conversation *ReportBug*. A *ReportBug* conversation starts with a dialog where a description of the bug is entered. It ends when the reporter is informed that the bug has been fixed. There can be an arbitrary number of reporters.

- There is a distinguished coordinator agent (a centralized software component implemented with the Microsoft Transaction Manager and Microsofts Active Server Pages). The coordinator is the performer for the *ReportBug* conversation but also a customer in two other conversations (*AssignBug* and *RemoveBug*).

- A distributor is a human agent responsible for the correctness of a particular software product. The *AssignBug* conversation is used by the coordinator agent to request the distributor to propose a programmer who may be able to remove the bug. There can be an arbitrary number of distributors.

- A programmer is a human agent that may be able to locate a bug and report successful removal of the bug back to the coordinator (conversation *RemoveBug*). There can be an arbitrary number of programmers.

The task of the coordinator agent is to implement each *ReportBug* conversation by a sequence of *AssignBug* and *RemoveBug* conversations. If a programmer is not able to remove a bug, the bug report is returned to the distributor who has to propose another programmer to solve the bug.

Each human agent has access to a list of the active conversations he is involved in. The list items are grouped by role and conversation specification and also indicate the currently active dialog step of each conversation. A human agent can switch freely between conversations and issue requests (if he is a customer) or create follow-up dialogs from a list of possible follow-up dialogs (if he is a performer).

This system has been implemented based on a generic subsystem for conversation management and dialog visualization. Since the main objective of the bug tracking software is to keep track of the state and of the history of problem-solving conversations in the enterprise, only very little code had to be written to implement the performer and customer rules of the coordinator. All synchronization tasks between the conversations have been successfully factored-out into library code.

Despite the fact that our process model described in Section 2.3 is based on purely sequential conversations, the implementation of agents (more precisely the implementation of the customer and performer rules) may introduce concurrency (by initiating multiple conversations) or synchronization (via mutexes, events, monitors etc.).

Figure 5 and 6 illustrate how agents (as process-oriented software components) can be composed systematically via conversation specifications using a Petri net formalism:

- In a first step, a state for each dialog in a conversation is created. The states are drawn at the borderline between two grey boxes. Each grey box corresponds to one of the participating agents. In Figure 5 there are three conversations with two resp. three dialogs.

- In a second step, a state is connected with its follow-up states via transitions based on the conversation specification.

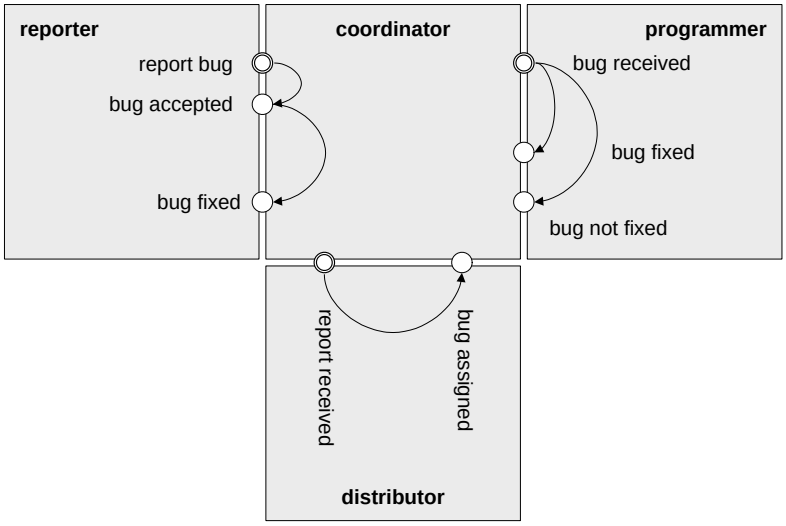

Fig. 5: Agents and Conversation in the Bug Tracker

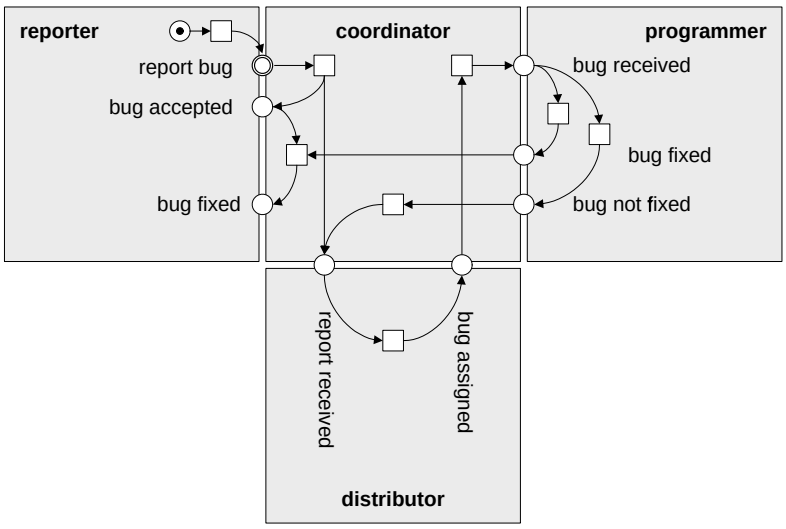

Fig. 6: Petri Net for combined Conversations of Bug Tracker

- In a third step, additional transitions and states (internal to an agent, but possibly connecting multiple of its roles) are added to formally define the desired interaction and synchronization between the agent's conversations. Since agents should be self-contained, autonomous software components, the only way to establish connections between roles of different agents are states corresponding to dialogs of conversations.

The resulting Petri net (cf. Figure 6) could be analyzed formally for deadlocks, safety and liveness. Alternatively, simulations could be carried out to detect mismatches between the design and the system requirements.

An interesting question of future research is to investigate whether it is possible to automatically generate rule implementations which only serve to connect (i.e. synchronize or coordinate) multiple conversation instances.

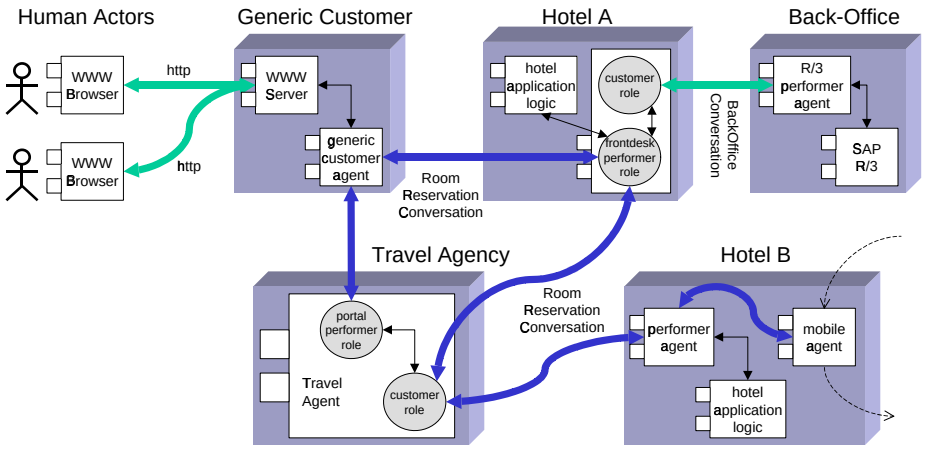

Fig. 7: Scenario for a Hotel Reservation System

3.2. A Broker for Hotel Reservations

The second example illustrates the use of process-oriented component specifications in a distributed environment that also supports conversations between *mobile and persistent agents* [25, 27]. The work described here is the result of a cooperation project with SAP AG which is interested in technologies and architectures suitable for the construction of scalable cooperative software architectures [32]. Our focus was also on developing a methodology applying commonly used UML notation and processes as basis, extended by our conversation model.

Starting with requirements analysis, use cases, actors and a glossary catching the key abstractions were created. On top of this, system interactions were defined by building a GUI mockup in form of HTML form pages. This leads us to the definition of both the conversation specifications (which could be automatically generated from the HTML forms) and a conceptual object model (which could be derived from the dialogs content model).

Diagram (b) in Figure 4 summarizes the agents and conversations in this particular scenario. The main agent developed in this project is a broker agent (e.g., hotel A) implementing a virtual hotel front desk. This front desk is a performer for a *RoomReservation* conversation that can be carried out via three different communication media, namely a HTML front-end for customers, message passing for remote agents at travel agencies and message passing for mobile agents that visit the front desk through the Internet. The front desk has to be able to use a (legacy) system like SAP R/3 as a back-office system to fulfill tasks as controlling, material management etc. which are not integral parts of room reservations.

The mobile agents and the travel agency agent may in turn act as *brokers* on behalf of customers contacting the agency via internet, e.g. to identify the cheapest offer available. The agent hotel B in Figures 4 (b) and 7 illustrates an important aspect of our process model: There can be multiple, possibly different agent implementations for the same conversation specification.

The deployment diagram in Figure 7 shows the scenario in detail. The component (or agent) of hotel A has two constituting roles: A performer role using the room reservation conversation specification to communicate with travel agencies or the generic customer mentioned below, and a customer role interacting with a back-office system using a "back office" conversation specification. The conversations are depicted as strong arrows. This back office may offer services not only for hotel A (not shown here). The travel agency has a similar topology of a combined customer role and performer role, which both use the same conversation specification.

Moreover, we developed a so-called "generic customer" application, which transparently and automatically visualizes conversations and dialogs (using HTTP) so that a human user can interact directly with any agent in the distributed system.

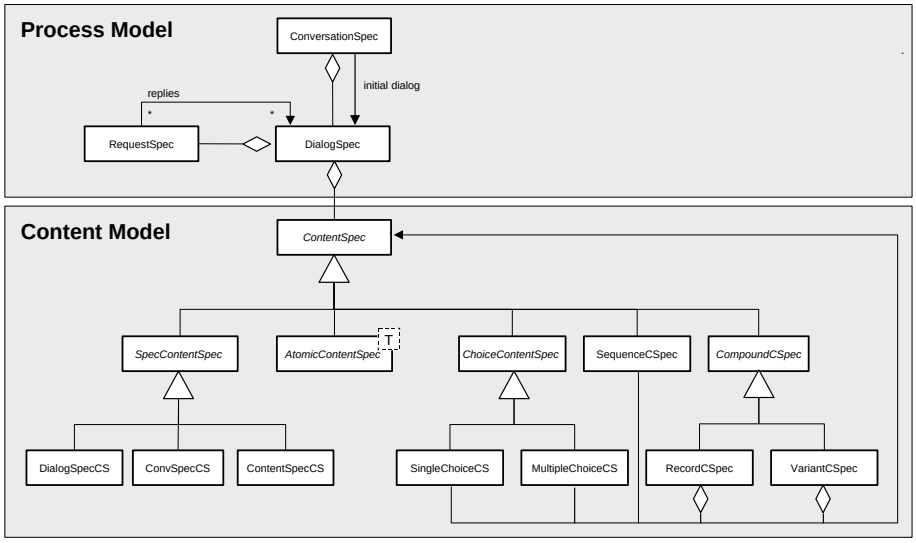

Fig. 8: An Object-Oriented Model for Conversation Specifications

A mobile agent may visit the different hotel systems and interact with the performer roles using the room reservation conversation specification to find the best offer.

The hotel front desk, the mobile agent and the SAP adapter are all implemented in the programming language Tycoon, a persistent programming language developed by our group. Each of these agents utilizes a common generic object-oriented class framework which provides means of defining (abstract) conversation-specifications, agents, roles, rules etc. [40]. This framework also supports conversations between persistent and mobile agents.

To summarize, this example shows how to develop a complex infrastructure of components based of roles with (possibly different) conversation specifications using UML notations as starting point.

4. BUILDING A GENERIC CONVERSATION MANAGEMENT FRAMEWORK

In this section, we briefly highlight the major steps necessary to build a generic conversation management framework. Details can be found in [40, 26, 20].

The first step is to define (persistent and mobile) representations for conversation specifications that match the process and content model of Figure 8. Therefore, we map conversation steps to dialogs. Requests are then used to connect them to form the overall process as a directed graph (see also Figure 9 below).

Information that is to be interchanged between customer and performer in a conversation is then assigned as content to the dialogs. Content may be complex, i.e. recursive and hierarchically structured. It can be constructed from atomic types like `boolean`, `integer`, `real`, `string`, `dateTime` and `currency`, and structured types like `records`, `variants`, `lists` and `single-/multiple-choices`.

Furtheron, content can be attributed with access rights (describing which actor may set or change a content's value) and with the author's signature (to certify the source of information). The specification of conversations, dialogs and content may also appear as the contents of conversations which is useful for higher-level conversations.

Using the visitor pattern, generators can be constructed to dynamically instantiate conversations, dialogs and contents (initialized with default values). For visualization and easy information exchange, the content specification is mapped to an XML DTD/schema, thus the conversion of dialogs to and from XML documents is straightforward [42, 44].

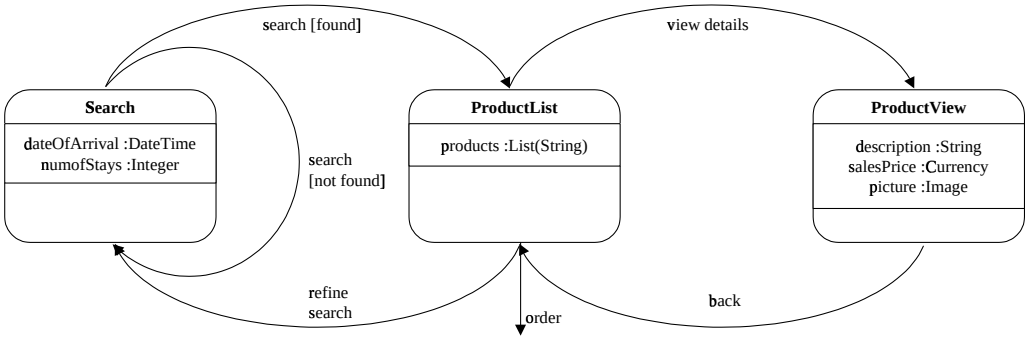

Fig. 9: A Detail from the *RoomReservation* Conversation Specification

The implementation of concurrent conversations and the synchronization between multiple concurrent conversations of a single agent is more intricate, in particular, if agents have to be persistent (i.e. outlive individual operating system processes) or if agents are allowed to migrate between address spaces while they are participating in conversations.

A detail from the *RoomReservation* conversation specification of the hotel reservation system is depicted as a state diagram in Figure 9. The `Search` dialog has two attributes denoting the date of arrival and the number of days the customer wants to stay at the hotel. The content of the dialog `ProductList` is a single-choice-list which contains the different products that match the customer's query. In the `ProductList` dialog, three requests can be issued by the customer: Return to the `Search` dialog to refine the search, move to the `ProductView` dialog to view details of a single product, or order the product selected from the product list.

The actual definition of an agent implementing this conversation as a performer looks as follows:

```
 (* create a hotel front-desk agent *)
IAgent agent = new Agent();

 (* create a performer role *)
IRole perfRole = new PerformerRole();
agent.addRole( perfRole );

 (* assign the RoomReservation conversation specification *)
perfRole.setConversationSpec( new RoomReservationCvSpec());

 (* add rules to the performer role *)
 (* dialog "
"Search" *)
perfRole.setRule("Search", "search", Search_SearchRule.class );

 (* dialog "ProductList" *)
perfRole.setRule("ProductList", "view details", PL_DetailsRule.class );
perfRole.setRule("ProductList", "refine search", PL_RefineRule.class );
perfRole.setRule("ProductList", "order", PL_OrderRule.class );

 (* dialog "ProductView" *)
perfRole.setRule("ProductView", "back", ProductView_BackRule.class );
```

A performer rule is simply a class with a method `transition()` which creates the follow-up (next) dialog and initializes the dialog's content. The next dialog has to be one of the dialogs specified in the conversation specification (this is verified by the state-enriched type-checker at

compile-time, see Section 5 below). All performer rules are subclasses of a common superclass `PerformerRule` which encapsulates rule management details.

As a concrete example, the class `PL_DetailsRule` contains the implementation of the performer rule for the dialog `ProductList` and for the request `view details`. Each transition method is executed in the context of an active conversation by a separate thread and may access the (typed) contents of the current dialog and the current request. The result of the transition method is the next dialog.

```
public class PL_DetailsRule extends PerformerRule {
  public void transition() {

     (* create the next dialog *)
    nextDialog("ProductView");

     (* fetch current product key from current dialog *)
    IProduct product = db.getProduct( lookup("dialog.products.current") );

     (* set attributes of next dialog "ProductView" *)
    lookup("next.description").set( product.getName());
    lookup("next.salesPrice").set( product.getPrice()
                                  * lookup("history.numofStays").get() );
    lookup("next.picture").set( product.getPicture());
  }
}
```

5. STATE-ENRICHED TYPE CHECKING

The information provided by the process model (i.e. the conversation specification) as well as type information from the content model (definition of dialogs content) can be used to support the implementor of the business rules by enforcing the correct enactment of these rules in the following two ways:

On the one hand, we have automatic consistency checking of rule implementations, which is described in Section 5.1. We have implemented this by means of a *state-enriched type-checker* [17] (which is based on the framework described in Section 4) using TYCOON-2, a persistent programming language developed by our group.

On the other hand, we claim that decoupling and flexibly re-connecting of processes while preserving the integrity of business rules can be achieved by analyzing the rules' code to find constraints related to both content and process definitions. Moreover, introducing also dynamic constraints which can be formulated at run-time permits even more flexibility [18]. This approach can be formally modeled by Petri nets and is presented in Section 5.2.

5.1. Consistency Checking of Rule Implementations

The state-enriched type-checker checks the business rules' code and provides support for the correct use of valid next dialogs (in performer rules) and valid next requests (in customer rules), depending on the current dialog (in both performer and customer rules) and the chosen request (in performer rules). Furthermore, the state-enriched type-checker ensures that only content of the chosen next dialog is assigned.

In the hotel reservation example, the performer rule attached to the *Search* dialog and *search* request could look as follows:

```
...
Enumeration products = db.getProducts( ... );

if ( products.hasMoreElements()) { // found some products

  // create ProductList dialog as next dialog
```

```
  nextDialog("ProductList");

  // set the ProductList dialog's content
  lookup("next.products").set( products );
}
else { // no products found

  // create Search dialog as next dialog
  nextDialog("Search");

  // set the Search dialog's content
  // use as search expression the previously entered search expression
  lookup("next.dateOfArrival").set( lookup("dialog.dateOfArrival").get());
  ...
}
...
```

The state-enriched type-checker ensures that only valid next dialogs (*ProductList* and *Search*) are created. It also enforces in case the *ProductList* dialog is created that only corresponding content (products) is assigned in the code following the `nextDialog()` statement. Similar conditions apply to the *Search* dialog.

Second, the state-enriched type-checker also provides support for type-safe access to the current dialog's content via the keyword `dialog` and, in performer rules, access to the content of the next dialog using the keyword `next`. Taken from the code fragment above, the code for setting the products

```
...
lookup("next.products").set( products );
...
```

is parsed and the type-checker inserts content specific type information so that type mismatches can be checked by the Java compiler:

```
...
((ListContent)lookup("next.products")).set( products );
...
```

In this case, the state-enriched type-checker will determine that the content-variable `products` is of type `ListContent`, and will change the code accordingly. The `ListContent's set()` method only allows an enumeration as parameter, thus type-safe access is enforced.

Finally, the state-enriched type-checker provides support for type-safe access to the content of earlier processed dialogs via the keyword `history`. This requires non-trivial control-flow analysis of conversation specifications to detect which dialogs are guaranteed to, cannot or may appear in the conversation history, and thus decide whether their contents are defined [10, 11].

Assuming the current dialog is *ProductView*, it must be preceded by at least one instance of the *Search* dialog (cf. Figure 9). Thus, the `dateOfArrival` content is always defined in the dialog's history and can therefore be referenced. The following example is part of the rule that is attached to the *refine search* request, wherein the most recent value of `dateOfArrival` is used.

```
...
  // create Search dialog as next dialog
nextDialog( "Search" );

  // use as default the previously entered value of dateOfArrival
lookup("next.dateOfArrival").set(lookup("history.dateOfArrival").get());
...
```

One can also iterate over the values of a given content in the conversation's history (i.e. "travel back in time"). Further details can be found in [17].

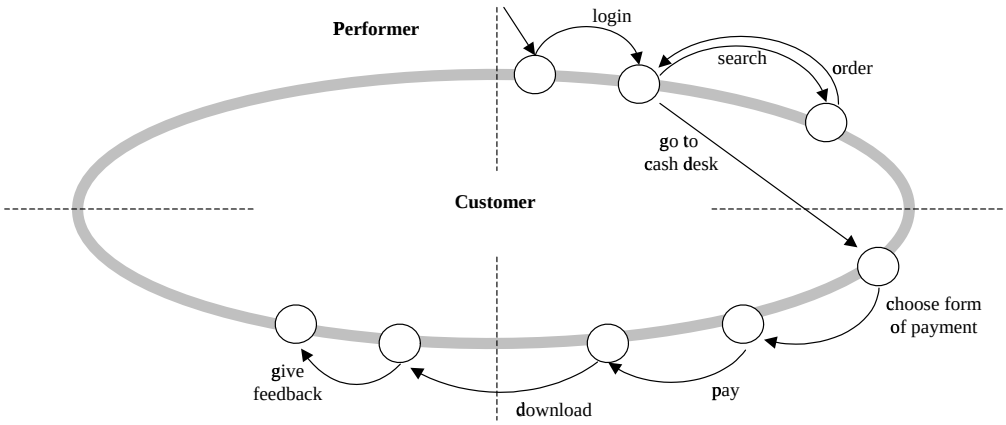

Fig. 10: State Diagram of the Overall Digital Goods Process

5.2. Constrained and Rule-based Composition of Conversation Specifications

To add flexibility to business processes, especially changing definitions of cross-organizational processes, is a common goal of many research groups. One approach favored in workflow management research is to introduce inheritance on workflow processes to adapt the agreed-upon process definitions in one organization without breaking consistency, e.g. by introducing deadlocks or livelocks [3].

While these ideas rely on subclassing the process definitions, we claim that taking process definitions as well as content specifications into account can provide a basis for non-subclass flexibility, i.e. reordering or regrouping of conversation specifications.

The state-enriched type-checker could be used to reorder conversation specifications that still can be checked for consistency at compile time.

Also, it adds support for decomposing conversation specifications into loosely coupled process fragments, i.e. subconversation specifications. It helps to create and rework conversation specifications in short time while still checking them for correctness, thus addressing business' needs for quickly adapting to changing requirements and workflows.

We illustrate our ideas by a Digital Goods internet shop example (see Figure 10). A customer enters the shop. He can search for products and place them in his shopping cart. This can be done several times, until the customer finally decides to go to the cash desk. If it is for the first time, he chooses a form of payment and a password. After that, he actually pays for the products (i.e. money is drawn from his bank account) and he downloads them (as they are digital). After using the products for a while, the customer can give a feedback. The process stops after the feedback. If a customer entering the shop is re-identified at the beginning of the process, he is asked to enter his password, thus avoiding to enter all customer-specific information again.

As the customer / performer interaction does not comprise parallel activities, the process can easily be modeled as a nondeterministic finite automaton. Furthermore, we model the Order process according to the speech act phases, see Section 2.3.

We identify three subconversation specifications *Login*, *Pay* and *Download* that we decouple from the main *Order* conversation specification and specify then as subprocesses. We intentionally leave out the connections between the main and the subprocesses, see Figure 11.

In the next step, the processes are refined as Petri nets. Customer and performer have activities on the inside resp. outside of the conversation cycle, where large places and transitions denote the customer's activities and the small ones denote the performer's (see Figure 12). The diagrams do not depict control flow, but only document flow, e.g. while the customer is handling the document, the performer process is not stateless. Notice the letter I in the first transition denoting the *initial rule* and the F denoting the *final rule*.

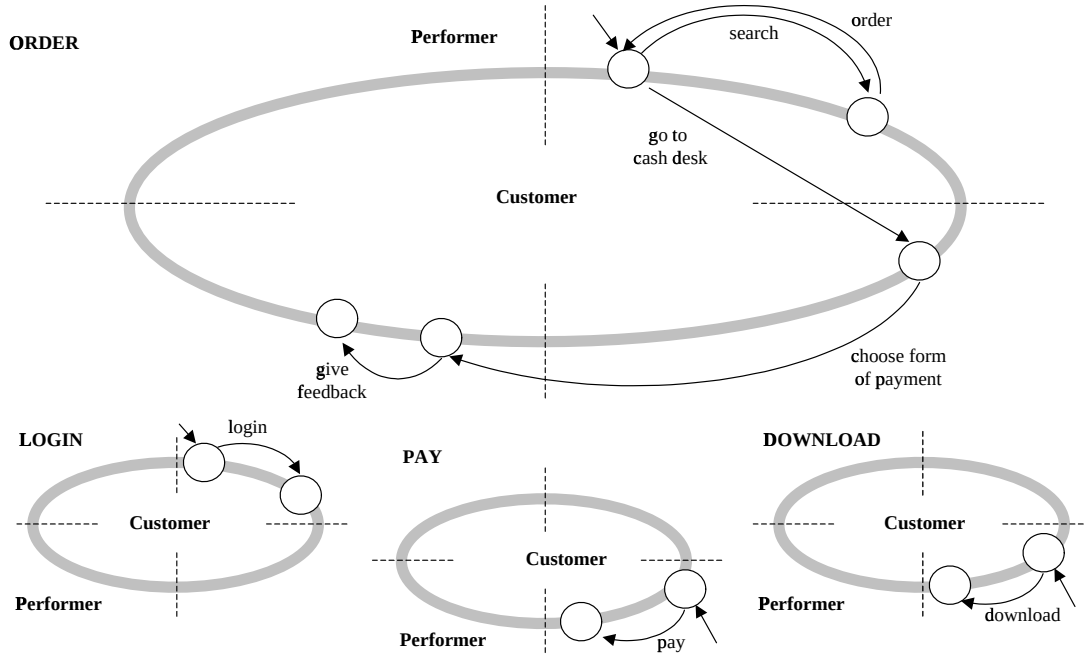

Fig. 11: Decoupled State Diagrams for the Processes

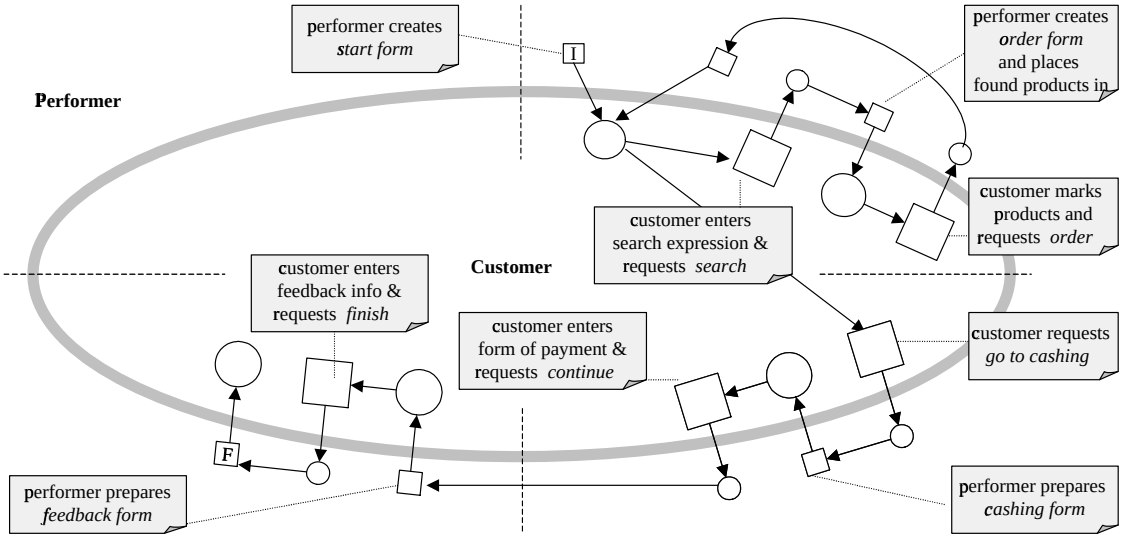

Fig. 12: *Order* Process Refined as Petri Net

The *Login*, *Pay* and *Download* subprocesses, which are depicted in separate Petri nets (cf. Figure 13), are decoupled from the *Order* process. They can easily be reworked and also be reused in different processes.

We now define pre- and post-conditions on the processes that state what content a conversation specification requires to execute and what content it will provide when terminated successfully. The following grammar defines possible conditions.

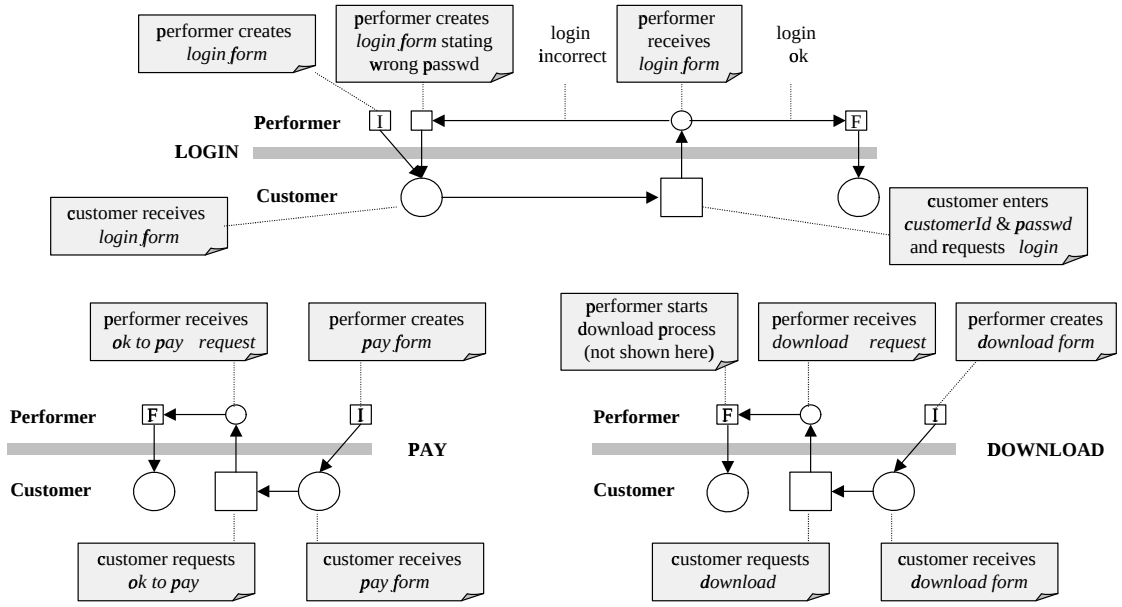

Fig. 13: *Login*, *Pay* and *Download* Processes Refined as Petri Nets

```
ContentIdentifier      ::= <String>
DialogIdentifier       ::= <String>
ConversationIdentifier ::= <String>

ContentCondition       ::= "defined(" ContentIdentifier ")"

ProcessCondition       ::= "done(" DialogIdentifier|ConversationIdentifier ")"

Condition              ::= ContentCondition | ProcessCondition |
                           Condition ∧ Condition | Condition ∨ Condition

PreCondition           ::= "pre(" Condition ")"
PostCondition          ::= "post(" Condition ")"
```

The glue connecting the processes is matching the pre- and post-conditions of conversations specifications and/or dialog specifications.

As an example, the *Order* specification may require some previously defined and used content named `customerIdentification`. This requirement can e.g. come from (performer) business rules that require a content `history.customerIdentification`. The performer's underlying enactment framework will check that it is a requirement that is not comprised by the *Order* process and thus will try to resolve the precondition. If this content is already provided, no precondition issues arise. If not, the enactment system will search for conversation specifications that have it as post-condition. The *Login* process provides the content `customerIdentification`. Therefore, the performer's enactment system will start the *Login* specification before enacting the *Order* specification. If the *Login* conversation succeeds, the content `customerIdentification` is henceforth defined. If the conversation fails, the content is not defined and thus the framework must retry to resolve the pending precondition.

The pre-condition of the feedback form dialog is the (successful) termination of both *Pay* and *Download*. These specifications have as pre-conditions that some customer provided information that is part of the cashing form dialog's content has been entered. Note that *Pay* and *Download* do not have interdependencies.

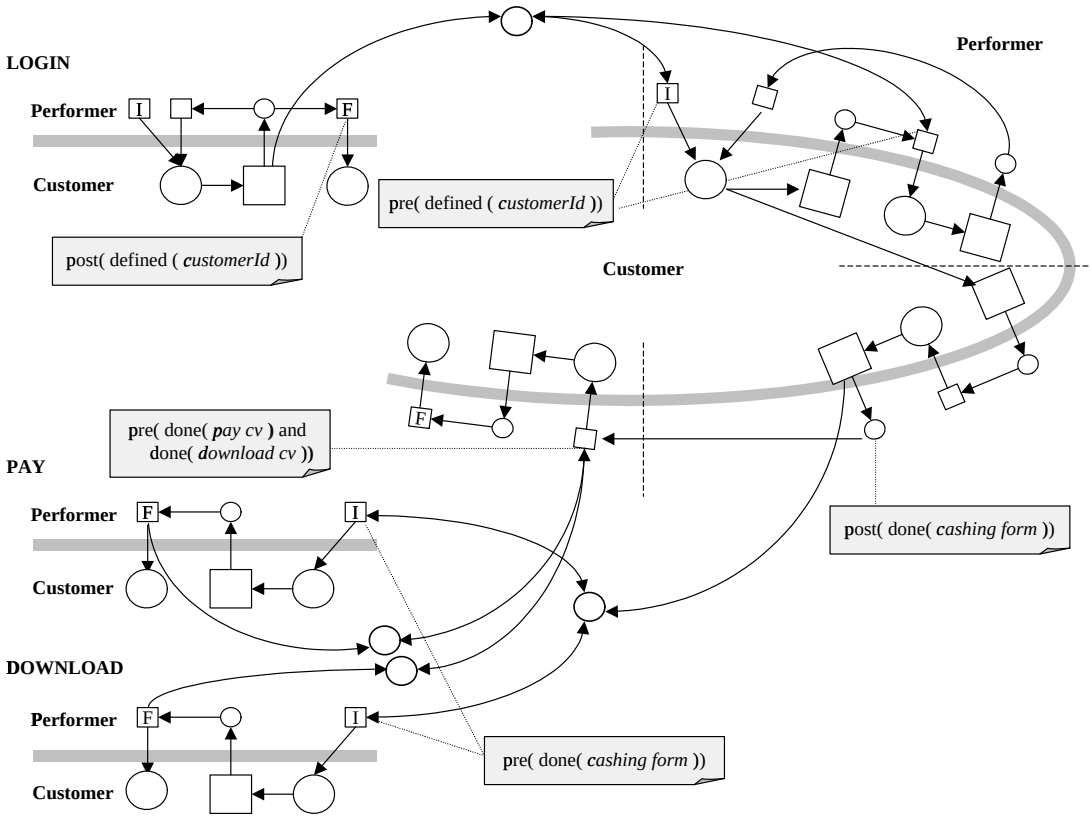

Fig. 14: Static Pre/Post Conditions in Petri Nets with Resolution

Pre- and post-conditions described here are static, because they result from analyzing the conversation specification and the content references according to the grammar in business rule code. Resolution of pre- and post-conditions is straightforward and can be done in Petri nets fairly easy by adding places for the conditions. Both the static conditions and their resolution in a Petri net are shown in Figure 14.

It is also useful to add and remove pre-conditions *dynamically* at runtime, while a conversation is running. For example, it can be reasonable to distinguish three kinds of customers and assigning different orders of *Pay* and *Download*:

- normal customers, who should always pay before download,
- business partners as customers, who can download and pay in parallel or
- special customers who can first download and pay later.

In the business rule code, dynamic pre-conditions can be added and removed. The following code shows an example:

```
...
if ( ... )
  SET.addPreCondition( payCvSpec, downloadCvSpec );
else if ( ... )
  SET.addPreCondition( downloadCvSpec, payCvSpec );
else // no condition means in parallel
...
```

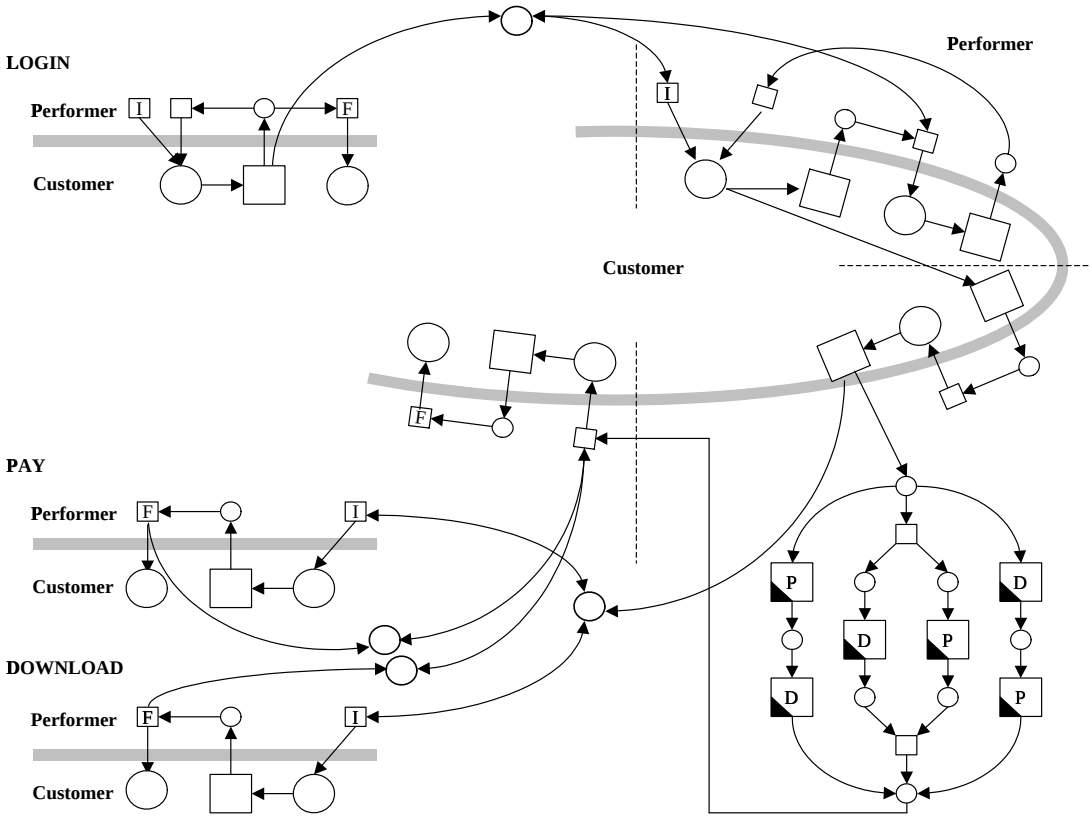

Fig. 15: Petri Net Including Static and Dynamic Pre and Post Conditions

Because dynamic pre-conditions can cause circular references and deadlocks just like static pre-conditions, it is necessary that the state-enriched type-checker checks them, too. This is done by control-flow analysis of the business rules. The identified dynamic pre-conditions are added to the process model (see Figure 15) and the resulting conversation specification can be checked for deadlocks.

All these tools are currently being developed [18] as a generic Java-Framework similar to and (conceptually) based on those described in [40, 17] and in Section 4.

6. RELATED WORK

Our model is conceptually based on speech acts [6] and the Conversation for Action (CfA) approach [41]. In AI, a lot of research has been done in the field of *autonomous agents* that rely on speech acts. Agents have been successfully used as the building blocks for business application infrastructures [13]. Many agent models use KQML [12, 22] for interaction and communication, e.g. IBM Aglets [24, 30, 23], Jackal [4] or JAFMAS [7].

However, these models and their implementations often lack a model for contents being transferred between agents. Instead they hold a knowlege base to support dynamic decision making. We believe that having a content model is a necessity for supporting business objects in information systems. Therefore, the actors in our model define a content model and no knowledge base, but a rule based decision strategy which must conform to a given conversation specification.

From a business information systems' point of view, the most important system specifications are intra- and inter-organisation process definitions [2]. Many workflow management systems support the design and enactment of such processes, e.g. METEOR [37]. Some of them are founded

on Petri nets, e.g. COSA [38], and can thus analyze the processes for properties like lifeness, deadlocks etc., e.g. WOFLAN [16]. Workflow management systems have a main focus on the design of processes, rather than on specifying the processes' contents. Mostly, documents are processed and exchanged without deeper knowledge of their contents modeled in the system. Opposed to workflow management systems, *groupware systems* like Lotus Notes allow for better document content modeling, but only support ad-hoc processes. Our approach brings together process and content models.

As mentioned in Section 5, one common approach for adding flexibility to business processes is to allow solely process changes that follow inheritance constraints [3]. Our approach is that taking into account process definitions as well as content specifications, we can provide an extended basis for non-subclass flexibility, i.e. reordering or regrouping of conversation specifications. The comparison between the two approaches shows that evolutionary changes to conversations that are based on inheritance allow for transferring running conversations to new specifications and thus avoid the problem of managing numerous variants of the same conversation [1], whereas changes to specifications that are not restricted to subclass relations – which is, in our experience, a frequent case – lead to versioning problems. This is especially true for long term conversations and persistent object systems.

Business process modeling following the common USDP (Unified Software Development Process Modeling) methodology [19] forces splitting processes into several unrelated use cases. The development process starts then from those use cases. Following strictly the USDP, the overall business process is not part of the modeling. The main drawback is that content-related dependencies between those use-cases (i.e. the different parts of the business process) cannot be checked. Our model permits the evolutionary development of the entire business process by means of the state-enriched typechecker as a tool for checking consistency.

In software engineering, components are being used to implement business objects [15]. However, these components and classes lack a formal description of their (process) semantics, which the commonly used interface and component definition languages (IDL, CDL) cannot provide for [21]. Our model maps business objects to both dialogs' contents (documents) and requests and thus extends the business objects by a process model. Dialog and request specifications form process and content definitions, i.e. conversation specifications, which are used for negotiating and enacting services by actors. These actors are software components whose interfaces are defined in terms of conversation specifications thus relating process and content.

7. CONCLUDING REMARKS

In this paper, we described a *business conversations* model which is based on a process-oriented and content-based perspective on software components. We illustrated the use of this model for the construction of distributed cooperative information systems using two practical examples from different application domains and explained how such software components can be implemented in different technologies using generic conversation management frameworks based on and extending standard object oriented system development models.

We furthermore illustrated in detail the steps to build a Digital Goods Shop performer actor, where we showed that encompassing flexibility in designing and redesigning processes can be achieved via the use of state-enriched type-checking. Research is towards further flexibility in components design, where flexibility does not affect the correctness of the enacted conversation specifications.

A necessary condition for process-oriented component specifications to become practically relevant is the availability of a widely accepted syntax / language to define and to exchange such specifications in distributed heterogeneous environments. XML as an emerging standard can be used as a common basis to start from, and build on top of XML DTDs / schemas conversation specification exchange formats [42, 44] or integrate with upcoming ones, e.g. the ICE standard [43].

REFERENCES

[1] W.M.P. van der Aalst. How to handle dynamic change and capture management information? An approach based on generic workflow models. Technical Report UGA-CS-TR-99-01, University of Georgia, Department of Computer Science, Athens, USA (1999).

[2] W.M.P. van der Aalst. Process-oriented architectures for electronic commerce and interorganizational workflow. *Information Systems*, **24**(8):639–671 (1999).

[3] W.M.P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. Computing Science Reports 99/06, Eindhoven University of Technology, Eindhoven (1999).

[4] IBM Alphaworks. Jackal – A Java-based communications infrastructure for agents and multi-agents. http://www.alphaworks.ibm.com/tech.

[5] Active server pages. http://www.activeserverpages.com (1998).

[6] J. Austin. How to do things with words. Technical report, Oxford University Press, Oxford (1962).

[7] Deepika Chauhan. JAFMAS: A java-based agent framework for multiagent systems development and implementation. Master's thesis, ECECS Department, University of Cincinnati (1997).

[8] Giorgio De Michelis, Eric Dubois, Matthias Jarke, Florian Matthes, John Mylopoulos, Mike Papazoglou, Klaus Pohl, Joachim Schmidt, Carson Woo, and Eric Yu. Cooperative information systems: A manifesto. In Mike P. Papazoglou and Gunther Schlageter, editors, *Cooperative Information System: Trends and Directions*. Academic Press (1997).

[9] Giorgio De Michelis, Eric Dubois, Matthias Jarke, Florian Matthes, John Mylopoulos, Joachim W. Schmidt, Carson Woo, and Eric Yu. A three-faceted view of information systems. *Communications of the ACM*, **41**(12):64–70 (1998).

[10] Javier Esparza. Model checking of persistent petri nets. Technical report, University of Hildesheim, Germany (1991).

[11] Javier Esparza. Model checking using net unfoldings. Technical report, University of Hildesheim, Germany (1992).

[12] Tim Finin et al. Specification of the KQML Agent-Communication Language – plus example agent policies and architectures (1993).

[13] Tim Finin and Yannis Labrou. UMBC AgentWeb. http://agents.umbc.edu/.

[14] F. Flores, M. Graves, B. Hartfield, and T. Winograd. Computer systems and the design of organizational interaction. *ACM Transactions on Office Information Systems*, **6**(2):153–172 (1988).

[15] Frank Griffel. *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt.verlag (1998).

[16] D. Hauschildt, H.M.W. Verbeek, and W.M.P. van der Aalst. Woflan: A petri-net-based workflow analyzer. Computing Science Reports 97/12, Eindhoven University of Technology, Eindhoven (1997).

[17] Patrick Hupe. A type system for analysis of dialog-oriented workflows in cooperative information systems. Master's thesis, Department of Computer Science, Hamburg University, Germany (1998).

[18] Patrick Hupe. A data- and process-oriented architecture for integration of cooperating information systems. Master's thesis, Department of Computer Science, Hamburg University, Germany (2000).

[19] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Publishing Company (1999).

[20] Nico Johannisson. An environment for mobile agents: Agent-oriented distributed databases. Master's thesis, Department of Computer Science, Hamburg University, Germany (1997).

[21] Detlef Kreuz. *Formal Semantics of Connectors*. PhD thesis, Hamburg University of Technology (1999).

[22] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250 (1997).

[23] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley Publishing Company (1998).

[24] Danny B. Lange and Daniel T. Chang. IBM Aglets Workbench. Programming mobile agents in Java: A white paper. Technical report, IBM Corporation (1996).

[25] B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. *Journal of Intelligent Information Systems*, **8**(2):167–191 (1997).

[26] F. Matthes. Business conversations: A high-level system model for agent coordination. In Sophie Cluet, editor, *Database Programming Languages: Proceeding of the 6th International workshop; proceedings / DBPL-6, Estes Park, Colorado, USA, August 18 - 20, 1997*. Springer-Verlag (1998).

[27] F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pp. 403–414, Santiago, Chile (1994).

[28] Florian Matthes. Mobile processes in cooperative information systems. In *Proceedings STJA'97 (Smalltalk und Java in Industrie und Ausbildung)*, Erfurt, Germany. Springer-Verlag (1997).

[29] Florian Matthes, Holm Wegner, and Patrick Hupe. A process-oriented approach to software component definition. In M. Jarke and A. Oberweis, editors, *Advanced Information Systems Engineering. Proceedings of the 11th International Conference, CAiSE'99, Heidelberg, Germany, June 14-18, 1999*, volume 1626 of *Lecture Notes in Computer Science*, pp. 26–40. Springer-Verlag (1999).

[30] Mitsuru Oshima and Guenther Karjoth. Aglets specification. Technical report, IBM Corporation (1997).

[31] Ingo Richtsmeier. A comparison of object- and agent-based communication. Master's thesis, Department of Computer Science, Hamburg University, Germany (1997).

[32] Volker Ripp. Reusing business process specifications. Master's thesis, Department of Computer Science, Hamburg University, Germany (1998).

[33] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Language Reference Manual.* Addison-Wesley Publishing Company (1999).

[34] SAP AG. BAPIs. http://www.sap-ag.de/bfw/interf/bapis/bapi.htm (1998).

[35] Gerald Schröder. *Cooperating Object Systems.* PhD thesis, Hamburg University of Technology, Germany (1999).

[36] J. Searle. Speech acts. Technical report, Cambridge University Press, Cambridge (1969).

[37] A. Sheth, K. Kochut, and J. Miller. METEOR project page. http://lsdis.cs.uga.edu/proj/meteor/meteor.html (1999).

[38] COSA Solutions. *COSA 2.0 User Manual* (1998).

[39] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, and P. Bernstein. Third-generation data base system manifesto. *ACM SIGMOD Record*, **19**(3):31–44 (1990).

[40] Holm Wegner. Object oriented design and implementation of an agent system for cooperative internet information systems. Master's thesis, Department of Computer Science, Hamburg University, Germany (1998).

[41] T.A. Winograd. A language/action perspective on the design of cooperative work. Technical Report No. STAN-CS-87-1158, Stanford University (1987).

[42] WWW Consortium (W3C). Extensible Markup Language (XML) Specification 1.0. http://www.w3.org/-TR/REC-xml (1998).

[43] WWW Consortium (W3C). The Information and Content Exchange (ICE) Protocol. http://www.w3.org/-TR/NOTE-ice (1998).

[44] WWW Consortium (W3C). XML Schema. http://www.w3.org/TR/xmlschema-1/, http://www.w3.org/-TR/xmlschema-2/ (1999).