A Process-Oriented Approach to Software Component Definition

Florian Matthes Holm Wegner Patrick Hupe

Software Systems Institute (STS)
Technical University Hamburg-Harburg
D-21071 Hamburg
{f.matthes,ho.wegner,pa.hupe}@tu-harburg.de

Abstract

Commercial software component models are frequently based on *object-oriented* concepts and terminology (e.g., classes, methods, messages, events) with appropriate binding, persistence and distribution support. In this paper, we argue that a *process-oriented* view on cooperating software components based on the concepts and terminology of a language/action perspective on cooperative work [Win87] (e.g., actors, roles, conversations, speech acts, conversation histories) provides a more suitable foundation for the analysis, design and implementation of software components in business applications.

We first explain the relationship between data-oriented, object-oriented and process-oriented component modeling and then illustrate our process-oriented approach to component definition using three case studies from projects with German software companies.

We also report on our experience gained in developing a class framework and a set of tools to assist in the systematic process-oriented development of business application components. This part of the paper also clarifies that a process-oriented perspective fits well with today's object-oriented language and system models.

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, 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 [DDJ+97, DDJ+98]

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 process-oriented approach to component definition using three case studies from projects with German software companies.

Details of the underlying process and system model are given in Sections 4 and 5. In these sections, we also report on our experience gained in developing a class framework and a set of tools to assist in the systematic process-oriented development of business application components. This part of the paper also clarifies that a process-oriented perspective fits well with today's object-oriented language and middleware models.

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 result 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 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.

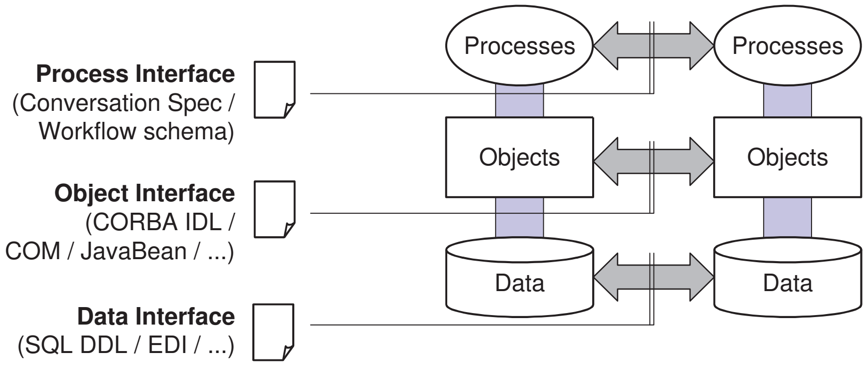

Figure 1: Three Approaches to Component Definition in Business Information Systems

For information stored in relational databases, SQL as the "intergalactic dataspeak" [SRL+90] provides both, a language for the exchange of vendor-independent data description (schemata, tables, views, ...) as well as a language for (remote) data access (via APIs like ODBC, JDBC, ...; 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.

The architecture of many non-relational information systems (also of "modern" systems like Lotus Notes, Microsoft Exchange and data warehouses) is also often described best as a collection of distributed data components.

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 classes, objects, (remote) references, attributes, methods, qualifiers, subclasses, interfaces, 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).

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

As exemplified by ODMG using CORBA IDL, an object-component model can also be used to describe "data-only" components, simply by omitting elaborate method and message specifications and only providing (set-oriented) get, set, insert and delete methods.

An object-oriented component interface definition defines 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.

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, shipment.replaceItem, shipment.deliver) 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.[1] Often it becomes necessary to restrict access of a specific client 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.
- In particular 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 relative to a specific client session.
- The lifetime of such execution contexts tends to be longer than the lifetime of the server process. Therefore, it becomes necessary to make such 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 (cyclic wait-for dependencies, race conditions, non-deterministic execution, ...).

As a solution to these problems we propose not to use object-oriented component interface definitions but process-oriented interface definitions between subsystems of a business information system following the language/action perspective on cooperative work [Win87, FGHW88, Sea69, Aus62]. For details on our model, see [Mat98, Mat97, Joh97, Weg98].

In a first step, we identify *actors* in a business information system. An actor can either be a human or an active process (thread, task, job, ...) 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.

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 `online shopping`. 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).

An actor can participate in multiple (possibly concurrent) conversations in 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 and available items dialog for the availability check). A dialog consists of a hierarchically structured *content specification* plus a set of *request specifications* valid in this particular

[1] For example, Microsoft Word 7.0 is a software component which has approximately one hundred interface classes.

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.). This set of requests is the request specification for the dialog.

For each request specification in a dialog there is a specification of the set of admissible follow-up dialogs. If this set contains more than one dialog, the performer can continue the conversation at run-time with 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.

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

Figure 2 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 *feedback*-phase gives the customer a chance to declare his/her satisfaction and 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 rolled-back.

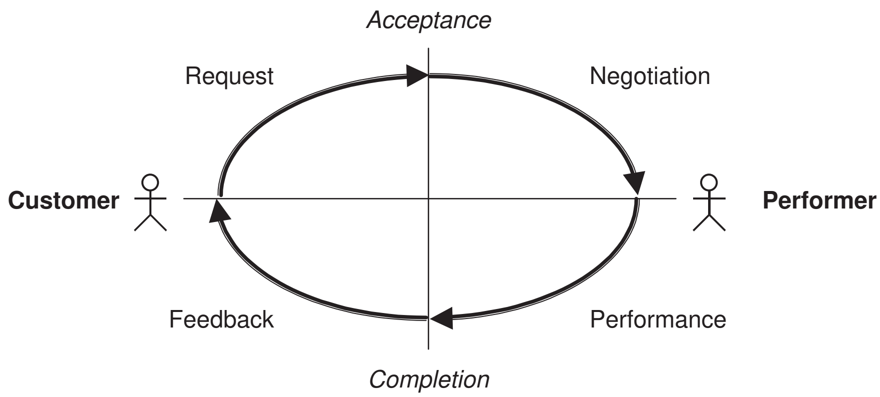

Figure 2: Phases of typical customer-oriented conversations

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 broken down into 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 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.

To summarize, conversation specifications include data specifications (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. For example, it is possible to access the history or the client of a conversation.
- It is possible to restrict requests 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 requests.

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.

3 Three Case Studies

In this section, we illustrate our process-oriented approach to component definition using three case studies from projects with German software companies, namely an internet shop implemented with Lotus Notes [Lot98], a bug tracker system implemented with Microsoft ASP [ASP98] and a hotel reservation system with an SAP R/3 backend using SAP BAPIs [SAP98].

The goal of these projects was to investigate whether the abstract process component model described in [Mat98, Mat97] and successfully implemented in persistent programming languages [Joh97, Weg98] is also a suitable basis for the implementation of process components using commercially relevant technology.

The conceptual basis for these projects is summarized in Table 1 that shows the (rough) correspondence between the abstract process component model concepts on the one hand side and the implementation concepts of the respective languages or systems used to systematically realize these concepts. For convenience, we also added a column describing the relationship between Java HTTP-Servlets and our model.

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.[2]

Figure 3 summarizes the agents and conversation specifications of the three case studies. In this diagram, an agent is indicated by a circle with an arrow. A conversation specification between

[2]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.

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

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

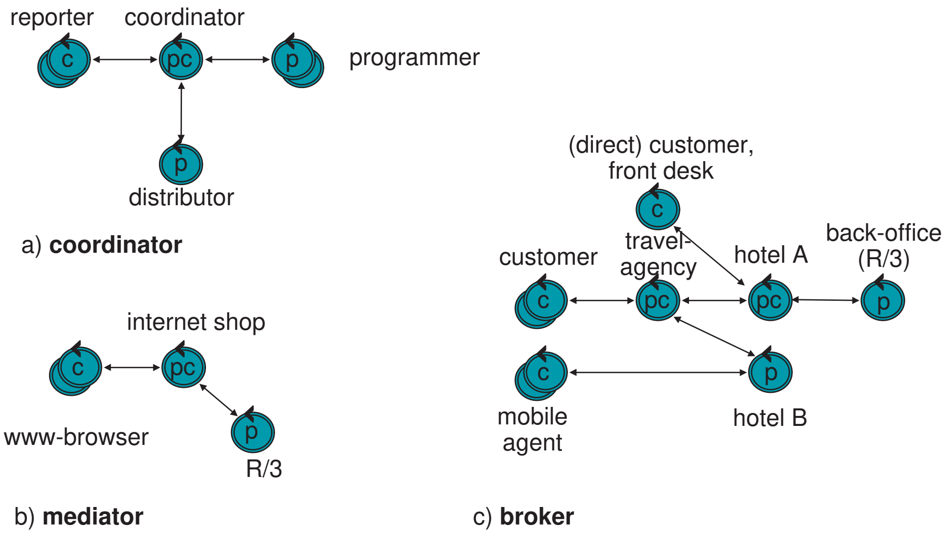

Figure 3: Agents and Conversations of Three Case Studies

two agents is indicated by a line with two arrows 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 3 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. At present we do not have a precise characterization of the constituents of an agent pattern.

3.1 Process-Oriented Modeling of an Internet Shop

In this example, an internet shop was created to support internet customers with HTTP clients and inhouse customers using Lotus Notes clients. The internet shop was implemented using a Lotus Notes Domino server. This implementation is based on a rather generic implementation of the agent model outlined in Section 2. This generic subsystem is responsible for managing multiple conversations (client sessions and server connections) and to decouple visual details (and language characteristics) of the user interface from the rule-based agent implementation. The server is also able to support persistent client sessions and to generate traces to obtain a feedback on the dialog sequences preferred by the customers.

Using this generic infrastructure, the system implementor first defines the conversation specification for `online shopping` including dialogs, requests and (most importantly) specifications for the admissible follow-up dialogs in each dialog step. The developer can utilize the standard Lotus Notes tools to design the dialogs and their content layout.

This initial system prototype is then validated interactively through the customer. If the customer selects a menu item or presses a button, he/she is prompted with a list of admissible follow-up dialogs to continue the conversation.

In a next step, the application developer implements rules (event handlers) for each request specification that appears in a dialog. This event handler at run time has to return a dialog object to be displayed to the user in the next process step. Each rule has access (via database variables) to the contents and requests of previous dialogs in the current conversation, a feature that is lacking badly in the “plain” Lotus Notes Domino environment.

The Notes Domino server also uses a SAP R/3 system for invoices, accounting and product availability checks. All client conversations share a common conversation with the SAP R/3 system which effectively serializes client dialog steps. To accommodate a larger number of customers, each client conversation could initiate its own conversation with R/3 or could use a pool of active conversations. This flexibility can be achieved by having conversations modeled as first-class (persistent) objects.

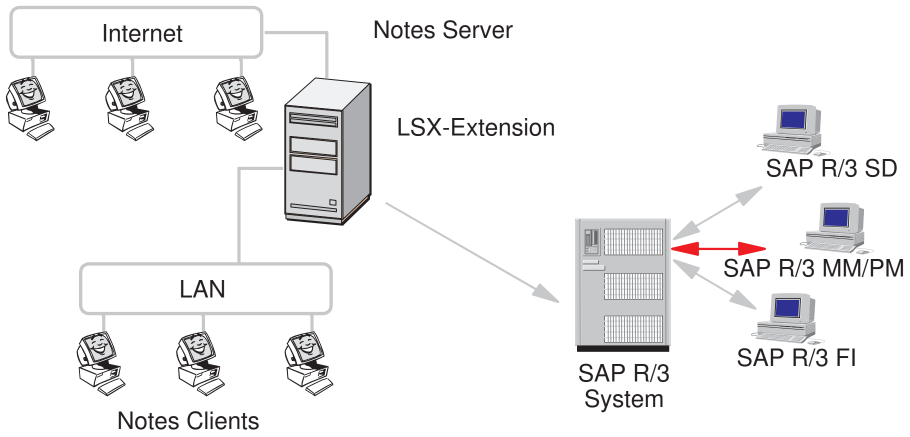

Figure 4: Process Component Interaction in an Internet Shop

3.2 A Workflow Manager for Software Bug Tracking

The next 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 3). 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. This conversation 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 system 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 component. 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.

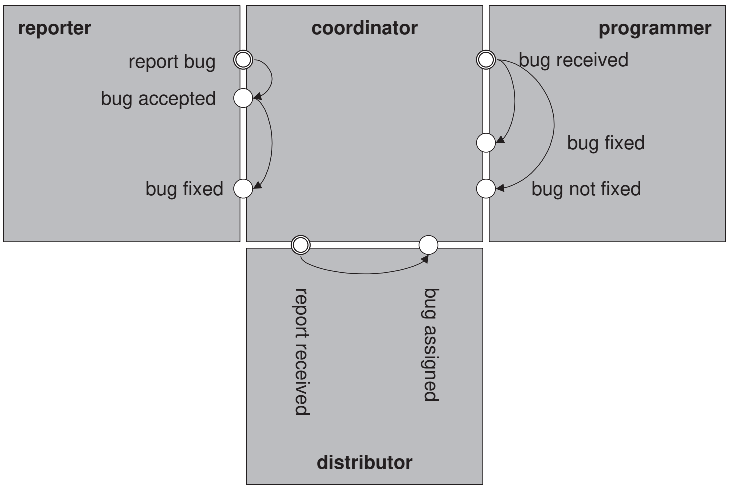

Figure 5: Agents and Conversation in the Bug Tracker

Each human agents 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).

Similar to the system described in the previous section, 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.).

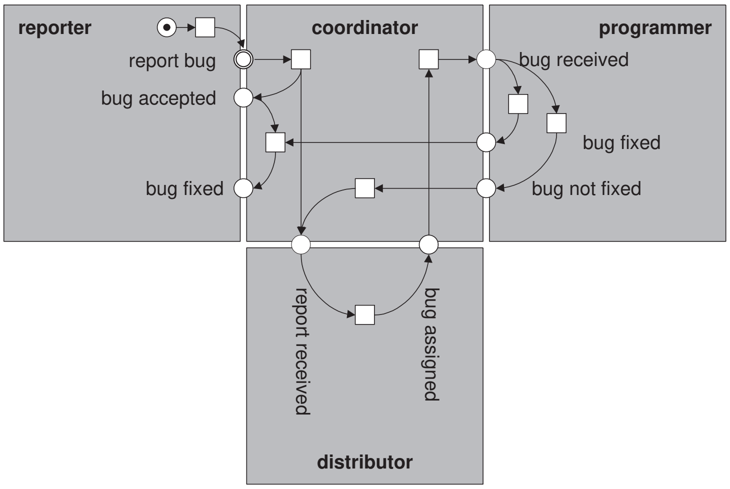

Figure 6: Petri Net for combined Conversations of Bug Tracker

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, one draws a state for each dialog in a conversation. 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, one connects a state with its follow-up states via transitions based on the conversation specification.
- 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 for 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.

3.3 A Broker for Hotel Reservations

The last example in this section illustrates the use of process-oriented component specifications in a distributed environment that also supports conversations between *mobile and persistent agents* [MMS97, MS94].

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 [Rip98].

Diagram (c) in Figure 3 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 do controlling, material management etc. which is not an integral part 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 3 (c) and 7 illustrates an important aspect of our process model: There can be multiple, possibly different agent implementations for the same conversation specification ($RoomReservation$ in our case).

Figure 7 shows some technical details of the implementations of the agents: 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.

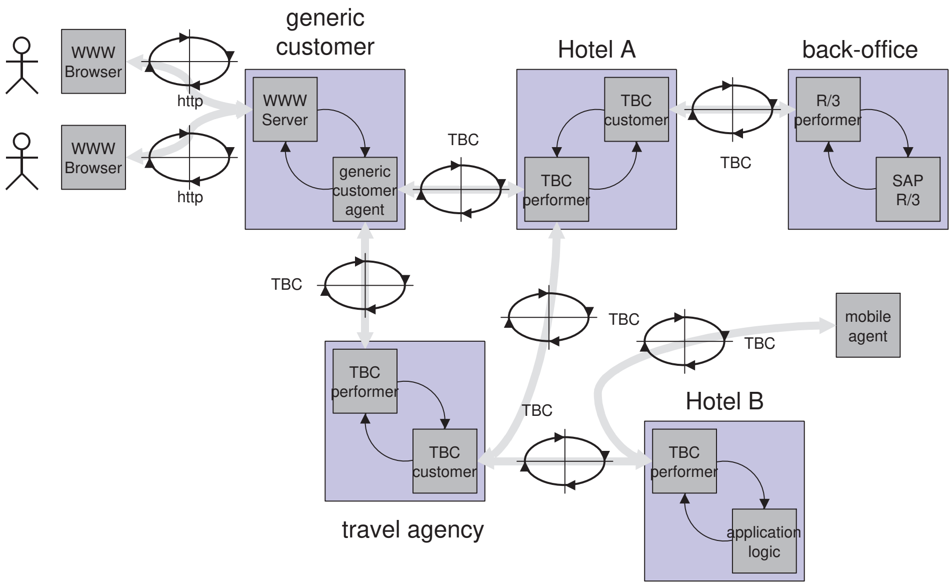

Figure 7: Scenario for a Hotel Reservation System

Each of these agents utilizes a common generic object-oriented class framework which provides means of defining (abstract) conversation-specifications, agents, roles, rules etc. [Weg98]. This framework also supports conversations between persistent and mobile agents.

Moreover, we developed on top of this framework 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. If desired, the generic customer can also be equipped with HTML files which provide templates to optimize the visual presentation of dialogs from individual conversations.

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 [Weg98, Mat98, Joh97].

The first step is to define (persistent and mobile) representations for conversation specifications that match the object model of Figure 8. The recursive definition of the class `Content Spec` supports complex dialog contents based on base types like `boolean`, `integer`, `real`, `string`, `date`, `time` and `currency`, and structured types like `records`, `variants`, `lists`, and `multiple-choices`. All of these constructs may be combined orthogonally. Finally, dialog specifications, conversation specifications and content specifications may appear as the contents of conversations which is useful for "higher-level" conversations.

In an object-oriented implementation language, this model maps directly into a class hierarchy. Using the visitor pattern, the construction of generators to dynamically instantiate conversations, dialogs and contents (initialized with default values), to visualize conversations, dialogs and contents and to transmit these objects is rather straightforward.

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.

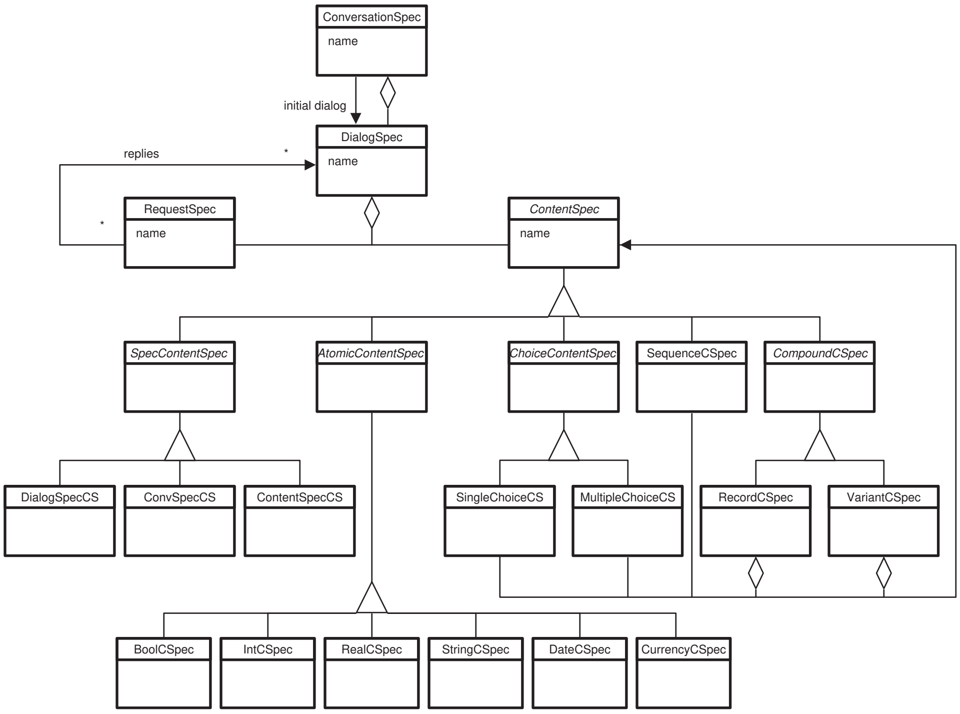

Figure 8: An Object-Oriented Model for Conversation Specifications

A detail from the *RoomReservation* conversation specification of the hotel reservation system is depicted as a state diagram in Figure 9. The `SearchAttributes` 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 `SearchAttributes` 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.

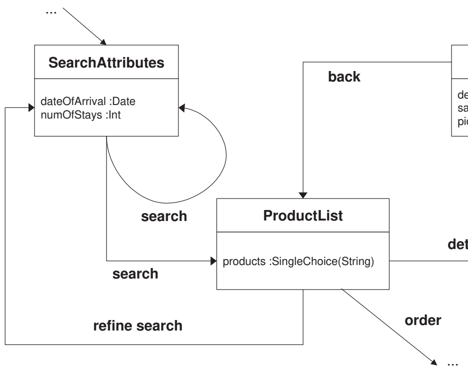

Figure 9: A Detail from the *RoomReservation* Conversation Specification

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

```
(* create a hotel front-desk agent *)
let agent :Agent = Agent.new("Hotel Front Desk"),

(* use existing RoomReservation specification *)
let convSpec :ConversationSpec =
    ConvSpecFactory.create("RoomReservation"),

(* create a performer role for this conversation specification *)
let perf :PerformerRole(PerformerContext) =
    PerformerRole.new( agent, convSpec ),

(* add rules to the performer *)

(* dialog "SearchAttributes *)
perf.addRule("SearchAttributes", "search", PerfRuleSearch.new ),

(* dialog "productList" *)
perf.addRule("ProductList", "details", PerfRuleDetails.new ),
perf.addRule("ProductList", "order", PerfRuleOrder.new ),
perf.addRule("ProductList", "refine search", PerfRuleRefineSearch.new ),

(* dialog "productView" *)
perf.addRule("ProductView", "back", PerfRuleBack.new ),
```

A performer rule is simply an object of a class with a method `transition()` which creates the follow-up dialog and initializes the dialog content. The follow-up dialog has to be one of the dialogs specified in the conversation specification (this is verified by the conversation manager at run-time). All performer rules are subclasses of a common superclass `PerformerRule` which encapsulates rule management details.

As a concrete example, the class `PerfRuleDetails` contains the implementation of the performer rule for the dialog `ProductList` and for the request `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 follow-up dialog.

```
class PerfRuleDetails
super PerformerRule(PerformerContext)
public methods

transition( conv :Conversation(PerformerContext),
            dialog :Dialog,
            request :Request ) :Dialog
{
    (* create and initialize the follow-dialog *)
    let next :Dialog = conv.newDialog( "ProductView" ),

    (* fetch current product key from current dialog "ProductList" *)
    let product :Product =
        lookupProduct( dialog.content["products"].singlechoice.current ),

    (* set attributes of next dialog "ProductView" *)
    next.content["description"].str := product.name,
    next.content["salesPrice"].currency :=
         product.price *
         conv.history[``SearchAttributes''].content.[``numOfStays''].int,
    next.content["picture"].image := product.picture,

    (* return the next dialog *)
    next
}
```

5 State-Enriched Type Checking

The rule implementation shown at the end of the previous section exhibits a significant potential for improvements exploiting the static knowledge already present in conversation specifications:

- Type-safe access to the contents of the current dialog should be supported. Thereby, spelling errors and also errors caused by schema changes can be detected at compile-time.
- The validity of the follow-dialog should be checked already at compile-time, exploiting the knowledge about the dynamic process structure of the conversation.
- Warnings should be issued if an implementor writes code to attach rules to requests which are not valid in a given dialog.
- Type-safe access to the dialog contents of earlier steps of a conversation should be supported. This requires a non-trivial control flow analysis of conversation specifications to detect which dialogs are guaranteed to appear in the history, which dialogs cannot appear in the history and which may appear in the history. We have formalized this decision procedure using temporal logic formulae [Hup98].

We have implemented these improvements by means of a so-called *state-enriched type checker* which is able to verify the consistency of rule implementations based on the additional knowledge of the dynamics of the system described by an object of class conversation specification. The state-enriched type checker makes use of a (typed) representation of the history and the current dialog.

As a consequence, the rule transition described in the previous section can be written in a more concise and type-safe way as follows:

```
class PerfRuleDetails
super PerformerRule(PerformerContext)
public methods

transition( cv :Conversation(PerformerContext),
            dialog :Dialog,
            request :Request ) :Dialog
{
    (* create the next dialog *)
    nextDialog( "ProductView" ),
    (* dialog is the current dialog's identifier *)
    (* next is the next dialog's identifier *)

    (* fetch current product from the current dialog "ProductList" *)
    let product = lookupProduct( dialog.products.current ),

    (* set attributes of next dialog "ProductView" *)
    next.description = product.description,
    next.salesPrice = product.price * history.numOfStays,
    next.picture = product.picture,

    (* return the next dialog *)
    next
}
```

The state-enriched type checker ensures that the dialog to be created in the `nextDialog` statement is valid. Similar checks are performed on requests generated in customer rules. The checker also ensures that variables referenced by the identifiers `dialog` or `next` are declared in the corresponding dialogs. In comparison to the former dynamically-typed code, no type information has to be specified; the dialogs' variables are completely statically typed.

One of the biggest advantages of state-enriched type checking becomes apparent when changing conversation specifications, e.g. when extending or refining an existing conversation specification by adding new paths, new dialogs or moving attributes (i.e. dialog content) from one dialog to another. Without typechecking, all rule code implementing the specification would have to be browsed and verified to check the compliance with the altered specification. In our example, the checker can verify that access to the variable `numOfStays` which is set in a previous dialog, is correct. All accesses to contents of recent dialogs are expressed by the identifier `history`.

6 Concluding Remarks

In this paper we described our *business conversations* model which is based on a process-oriented perspective on software components. We illustrated the use of this model for the construction of distributed cooperative information systems using three practical examples from different application domains and explained in quite some detail how such software components can be implemented in different technologies using generic conversation management frameworks.

The technology and formalization of state-enriched type checking may have other interesting application areas (e.g., verifying the consistency of a set of interacting Java Servlet implementations or generating strongly-typed access code to CGI-arguments).

A necessary condition for process-oriented component specifications to become practically relevant is the availability of a widely accepted syntax / language to write down and to exchange such specifications in distributed heterogeneous environments. One could either "abuse" CORBA IDL as a starting point for conversation specification or one could utilize SGML / XML documents that conform to an agreed-upon `ConversationSpecification` DTD.

References

[ASP98] Active server pages. http://www.activeserverpages.com, 1998.

[Aus62] J. Austin. How to do things with words. Technical report, Oxford University Press, Oxford, 1962.

[DDJ+97] 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.

[DDJ+98] 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, December 1998.

[FGHW88] 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.

[Hup98] Patrick Hupe. Ein Typsystem zur Analyse dialogorientierter Workflows in kooperativen Informationssystemen. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, November 1998.

[Joh97] Nico Johannisson. Eine Umgebung für mobile Agenten: Agentenbasierte verteilte Datenbanken am Beispiel der Kopplung autonomer "Internet Web Site Profiler". Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1997.

[Lot98] Lotus Development Internations Corporation. Lotus / Domino Server 4.6.1, 1998.

[Mat97] Florian Matthes. Mobile processes in cooperative information systems. In *Proceedings STJA'97 (Smalltalk und Java in Industrie und Ausbildung)*, Erfurt, Germany, September 1997. Springer-Verlag.

[Mat98] F. Matthes. Business conversations: A high-level system model for agent coordination. In *Database Programming Languages: Proceeding of the 6th International workshop; proceedings / DBPL-6, Estes Park, Colorado, USA, August 18 - 20, 1997*. Springer-Verlag, 1998.

[MMS97] B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. *Journal of Intelligent Information Systems*, 8(2):167–191, 1997.

[MS94] F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994. (An extended version of this text appeared as [MaSc94b]).

[Rip98] Volker Ripp. Verbesserung der Lokalität und Wiederverwendbarkeit von Geschäftsprozeßspezifikationen: Probleme und Lösungsansätze am Beispiel kundenorientierter Hotelgeschäftsprozesse. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, March 1998.

[SAP98] SAP AG. BAPIs. http://www.sap-ag.de/bfw/interf/bapis/bapi.htm, 1998.

[Sea69] J. Searle. Speech acts. Technical report, Cambridge University Press, Cambridge, 1969.

[SRL+90] 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, September 1990.

[Weg98] Holm Wegner. Objektorientierter Entwurf und Realisierung eines Agentensystems für kooperative Internet-Informationssysteme. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, May 1998.

[Win87] T.A. Winograd. A language/action perspective on the design of cooperative work. Technical Report No. STAN-CS-87-1158, Stanford University, May 1987.