2

The Form-Based System Paradigm

Enterprise systems encompass online transaction processing systems, enterprise resource planning systems, electronic data interchange, and e-commerce. This means the system class of interest can contain a small web shop as well as a huge system like the SABRE flight reservation system, which connected 59,000 travel agents in the year 2002 [155].

In this chapter we give an outline of our model for enterprise applications. An enterprise system can be seen as an installed and running enterprise application. The basic type of enterprise system we call a *unit system*: that is, a system which we consider as a single unit for our purposes. From the user's perspective a unit system is a black box. It is characterized by the interfaces through which it is accessed. Each unit system is a single unit of abstraction, it is a single *abstract data object*. The interface of a unit system which is directly accessible for us is the *human–computer interface* for interaction with the user. A unit system can also have an interface to other unit systems, and we will call this a *service interface* . Of the two kinds of interfaces of a unit system the human–computer interface is the more tangible one, therefore we begin our outline of the system modeling approach with this type of interface. In our method, the human–computer interface for communication with one unit system is session based, and we call it the *submit/response* style interface. One can conceive of another kind of interface which is sessionless and resembles a mail client with its mailbox. We will discuss such an interface designed for communication with multiple unit systems later in the book. But here we concentrate on the session-based interface type. It captures the key concepts behind several widespread interface types for enterprise applications, e.g., web interfaces. In one sentence one can say that the submit/response style interface models the human–computer interaction as an alternating exchange of messages between the user and the computer. But before we try to understand submit/response style systems in this way we look at them solely from the perspective of the user.

2.1 The Submit/Response Style Interface

We introduce the class of submit/response style interfaces by using a familiar application as an example, namely an online bookshop as can be found frequently in a similar form on the Web. Chapter 3 is devoted solely to an informal description of this example bookstore.

We have designed the following considerations in such a way that the reader can participate in the development of the ideas about the interface types. This is intended to be neither a historic line of development nor a necessary argument; it is just considered to be helpful, instructive, and easy to follow.

Submit/response style interfaces show at each point in time a page to the user, the *current page*. Two such pages, which are taken from our example bookstore, are shown in Fig. 2.1, i.e., a page showing the contents of the user's shopping cart and a page for gathering personal data.

Fig. 2.1. Example pages of the online bookshop

A submit/response style interface allows the user to perform two kinds of interactions with the interface: we call them *page edits* and *page change*. Page changes are singular interactions which change the page, i.e., the current page is replaced by a new page. Page edits are interactions with the current page, namely the filling out of a form or resetting a form. Forms are the only editable parts of the page, and are made of input elements. These input elements can be quite sophisticated by themselves. A very sophisticated form element is a text field that allows the input of formatted text, as can be found in some interface technologies.

There is a hierarchy in these two kinds of interaction. Take the search option as an example. First you enter keywords by page interaction. Then you press the search button and the system shows the page with the search results by performing a page change. The page edit is always a preparation for

the page change in this style. We call this the *two-staged interaction* paradigm of submit/response systems.

During the heyday of GUI-based client/server programming such interfaces were often considered as bare metal legacy technology. The advent of the web browser as a new thin client has shown many reasons why submit/response style interfaces are here to stay. On the one hand there are proven system architectures for submit/response style systems. Classical mainframe architectures like CICS are still in use and being constantly improved. Some ubiquitous commercial off-the-shelf (COTS) products are successful because they have a mature system architecture. They provide working solutions for enterprise applications, and they take into account the substantial non-functional requirements of enterprise applications. New vendor-neutral and platform-independent enterprise computing approaches like J2EE are emerging, targeting the same driving forces such as the classical approaches.

But submit/response style systems do not just have proven software architectures. Surprisingly, submit/response style interfaces can have cognitive advantages, too. This means that submit/response style interaction can foster usability in many cases, simply because it is often the natural solution with respect to an automated enterprise functionality.

2.1.1 Proven System Architecture for Submit/Response Style Systems

Enterprise applications are data-centric and transaction-based. The submit/response style interface is not tied to any specific technology. On the contrary, the same characteristics can be found in many technologies, e.g., HTML-based clients and mainframe terminals. Even the screens of a GUI-based COTS system follow the submit/response style interface metaphor.

An important class of systems with submit/response style interfaces are systems with ultra-thin clients, encompassing terminals and HTML browsers, see Fig. 2.2. Ultra-thin clients are used for creating an interface tier that does not contain business logic in itself. Ultra-thin clients cache the user interaction on one page in the client layer. The page sequence control logic – or workflow controller – is also not hosted by the client layer but rather by the server layer. Ultra-thin clients fit neatly into the transactional system architecture, be it one of the classical proposals [23, 130] or a more recent proposal [181]. Transactional system architectures successfully target many problems: system load, performance maintainability, scalability, security, and others.

The interaction with a system/response style system is a repeated alternation between data processing and the presentation of a new screen. The dialogue appears to the user as a sequence of editable screens: the dialogue steps are *screen transactions*. The presentation layer of a system is responsible for a preprocessing of data submitted by the user, the triggering of appropriate business rules, and the presentation of the correct new screen. Given a multi-tier system architecture, there is no requirement that this logic be hosted by
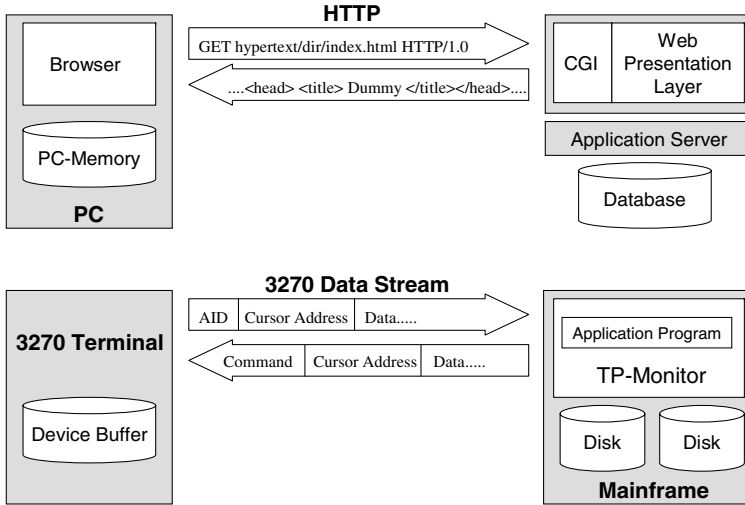
Fig. 2.2. Examples of ultra-thin client based submit/response style systems

the application server tier. In the SAP R/3 system [208], see Fig. 2.3, it is actually hosted by the client tier. The SAP R/3 system architecture is optimized with respect to the notion of commercial off-the-shelf software. In a full version of the SAP R/3 system the vertical architecture depicted in Fig. 2.3 is completed by a horizontal architecture consisting of a production system, a consolidation system, and a development system: the necessary customization of the system is only possible in a defined safe way by deploying new modules via a special transportation system.
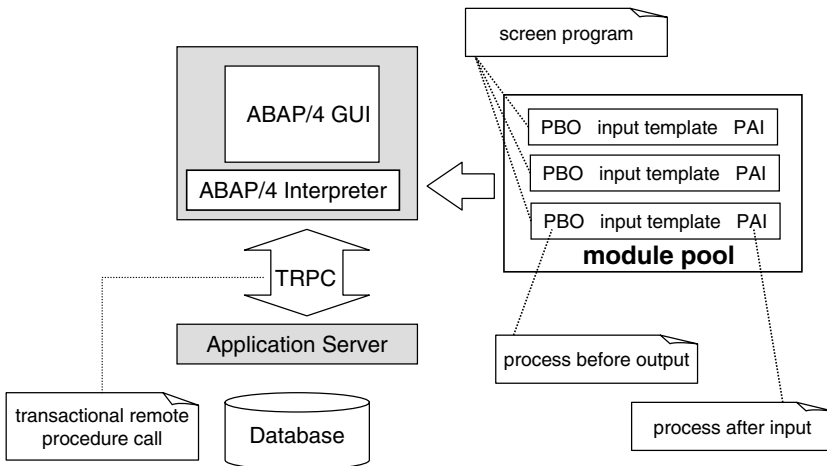

Fig. 2.3. SAP R/3 architecture – a client/server submit/response style system

2.1.2 Cognitive Advantages of Submit/Response Style Systems

Form-based interfaces have clear advantages for the self-explanatory character of a system. The usage of the system is intuitive, since it is guided by a paper form metaphor. However, the importance of the submission process is notable; therefore we want to characterize the metaphor as a *submission form metaphor*. The difference between temporary input and submission, or "sending," is intuitive and fosters the user's understanding of the system. The form-based metaphor has a multi-tier structure of its own, without being fixed to an implementation. The two classes of interactions structure the work of the user into the work-intensive frequent page interactions and the punctual and atomic interactions of the "serious" kind, namely the page changes which also happen to be the conclusion and separation of logically disjoint bunches of work. The submit/response style character puts the user in command of the timing of system usage. It protects the user from irritating disruption of his or her work by incoming information.

In form-based interfaces the submission of a form is an operation that has exactly the semantics indicated by the metaphor. In computer science terms we have compared the submission of an actual parameter list with a method name. The submission form metaphor views interaction with the system as filling out virtual paper forms and submitting them to a processing instance, which represents the core system.

The metaphor has the qualified name submission form metaphor, because other form interface types can be found as well. For example, desktop databases as found in office suites allow form style interfaces, which possess page navigation buttons. Input in this form immediately changes the model. We call such a form style interface a *formlike view*. Applications using formlike views are in principle required to have synchronous views of the data: if two formlike views currently show the same data, and the data are changed in one formlike view, then the other formlike view has to be immediately updated. Many implementations, however, have to stick to polling mechanisms, which leads to latency effects in the update process. Well-known and even worse examples are file managers, which recognize state changes frequently only after manual refresh. It is important to recognize that the necessary refresh in this case is a bad implementation, while the reload mechanism of submit/response style applications is a logically necessary feature.

In desktop databases the model state is the persistent state. Other applications with formlike views have non-persistent states, e.g., spreadsheets.

The submission form metaphor has the advantage of possessing a clear semantics. The two-staged state change due to the two-tiered model is an integral part of the metaphor. This is quite in contrast to, for example, the important desktop metaphor. Consider the important drag and drop feature, which is at the very heart of the desktop metaphor. Drag and drop means regularly either copy or move, and hence can lead to two different effects.

The submission form metaphor is accompanied by the *response page principle* for showing responses of the core system. The submission of a form is a page change, i.e., the page that hosted the form is hidden and a new page is shown. This new page is the response from the server. The response page has three important functions:

- Notifying the user of the immediate status of the submitted form.
- Showing new information to the user.
- Offering new interaction options.

The immediate status of the submitted form is the system's immediate response to the form. Depending on the business logic this may or may not be the completion of the form processing.

- Consider the entry of a new date in a web calendar tool. The response page is the new calendar view with a short notification message. The form has been completely processed.
- Consider the submission of an order in an online brokerage system. The response page is a notification of reception. The execution of the order, however, takes place asynchronously.
- Consider the submission of an e-mail in a mail account on the Web. The response page logically is only a notification of some overall validity of the submitted data, e.g., the recipient's address contains an at-sign. The completion of the intended effect, i.e., the delivery of the e-mail, is not acknowledged at all.

2.1.3 Semantics of Page Change

Each page can contain different forms. A form connects input elements to a page change option, the submit button. Of course the intuition is that only the page edit in the form that belongs to a submit button gives the intended meaning of this command.

This meaning is captured in a message-based model, which we use for the submit/response system. The user interface is considered as a distinct system, which we call the *conceptual session terminal*, or terminal for short. It is very much an abstraction of today's web browser used as a client for an enterprise application. As the name terminal suggests, the terminal is considered to be connected with the unit system, which means that it can communicate by messages and only by messages. The conceptual session terminal has a state, namely the current page shown to the user, optionally including some invisible information within that page, as well as the page edit the user has performed so far on this page. The page edit is kept in the terminal until the user performs a page change, which belongs to a form. Then the contents of the input elements of this form are transmitted as a data record to the unit system. The input elements of this form can contain the page edit of the user, or data which was pre-filled on the page, so-called default data. The

transmitted data record is tagged with the name of the form, and together this message is like a remote procedure call. The name of the form is something like the name of the procedure. It leads to an action on the unit system, and this action always produces a result, given as a page description. This page description is a message that is transmitted back to the terminal, and the described page replaces the current page the user has seen before. This gives in effect a page change. We call the new current page the *response page* to the submission, the received message the response message, and if we do not want to distinguish it, we call it the *response.* The page change is therefore the submission of a parameterized command, and the new page is the result. The terminal is locked between send and receive. The remote method call is therefore a synchronous procedure call. This alternation of submissions and responses has of course given rise to the term submit/response.

The parameter of the submission can of course be empty. There is only one type of page change. Each page change can, however, transmit data, which were not rendered on the page.

All page edits that have been performed on input elements which do not belong to the submitted form are therefore lost; the state of the conceptual session terminal after a page change is exclusively the response page.

Of course there are many other possible types of interfaces than the submit/response interface explained so far, e.g., interfaces which support several pages at once. However, the submit/response style system has its advantages in that it is quite expressive yet simple and primarily it is very regular. The strict alternation between the user and system messages yields many advantages for modeling. Therefore it is very suitable for the high abstraction level, on which we want to focus during analysis. A key concept here, which contributes to the whole method's characteristics, is the notion that the user can submit a whole compound data object with each message.

2.1.4 Dialogue Types

We have explained submit/response style interaction as the alternating interchange of messages. We now want to introduce static typing to these messages, and this step alone will lead to a plethora of interesting new properties of our interfaces.

First we want to introduce static types for the response pages. This means that only a finite number of page types are allowed for each interface. It allows us to give a natural yet rigorous meaning to the finite number of pages depicted in screenshot diagrams (sometimes called non-executable GUI prototypes); they simply represent the page types. Furthermore it allows us to give precise semantics to the arrows in these diagrams representing possible page change in the following way.

The current page has a type from a finite number of page types. We conceive the type of the current page as a *finite state aspect* of the terminal. (A finite state aspect is a reduction of the state of a system, which is of interest

for the modeler. This is known from finite state modeling in many domains. Consider the finite states a process can have in an operating system. Of course each process can in principle have infinite states, but the finite states are the reduction of interest for the modeler.) The terminal can then be seen as a finite state machine. The arrows are naturally characterized as transitions.

We now turn to the user messages. They are statically typed as well. Therefore there can be only a finite number of possible user messages. Each form on a page must be assigned a single user message type. The page edit on this form prepares an instance of this type. The page change is then used for sending this instance as a method parameter. We identify the concept or the type with the concept of the procedure name. Therefore the type of the message already determines the processing action of the unit system. We call this procedure of the unit system the *server action*. A form on a page is therefore an editable message instance.

For each page type the number of page changes is constant or bound by a constant. Consider a catalogue page which contains a list of books. Each book can be put into the shopping cart with a single click. If we model these interaction options as separate page changes, then the number of page changes is not bound by a constant. We therefore conceive all these interaction options as addressing the same page change, but providing a different parameter every time. In this way the list of interaction options forms a single conceptual interaction.

2.1.5 Conditional System Response

If a message is sent to the unit system, the system's response is conditional, depending on the message and on the system's internal state. Of course the system's response is conditional with respect to the content of the page, e.g., in the case of selecting a book, the shopping cart as the system's response depends on the previous cart state as well as on the chosen book. But the system's response can be conditional with respect to the page type as well. Take a system login dialogue as an example. The business logic says that if a user has never bought anything, then after six months the username will be deactivated and can be taken by another user. The submission of username and password can therefore have a number of different effects.

- If the username belongs to a valid account and the password is valid, then the welcome screen for registered users is shown.
- If the username belongs to a deactivated account and the password is valid, then the user gets a screen informing him or her that the account has been reactivated.
- If the username belongs to a valid account which has been taken over by a user, and the password is the last password of an old user, then the user gets a notification that his or her account has been collected and redistributed. A new account is offered to him or her.

- If the password is invalid, then the user is informed about that, and he or she is offered assistance for forgotten passwords.

The page type as a state of the finite state machine can therefore end up in four different pages triggered by the same page change. We want to have a model that captures both the fact that the user has chosen one single page change as well as the correct page type of the system's response. For this purpose we use a novel bipartite state model. The rationale is that we model the system's processing of the request as a separate state. Therefore if the user triggers a page change, the finite state model of the terminal changes into a processing state, which we call *server action*, and depending on the received response page the state model changes into the respective page type. The state of the terminal therefore alternates between page types and server actions. The server actions are left automatically as soon as the unit system's response is received. In the same way as we identify the page state with the type of the displayed message, we now identify the server action with the type of the message that has been sent from the user to the system and which is processed during this server action. The resulting bipartite state machine is painted as a *formchart*.
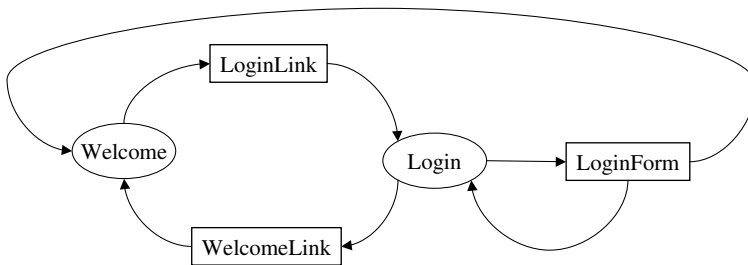

Fig. 2.4. Example formchart for a system login capability

Fig. 2.4 shows a first motivating example formchart for a system login capability. From the system welcome page it is possible to reach a login page via a link. From there it is possible to abort the login dialogue via a link back to the welcome page. Of course the login page offers a login form. The response of the server action representing the login form is conditional: if an error occurs the dialogue flow is branched back to the login page reporting an appropriate error message, otherwise it is branched to the welcome page. The formchart snippet in Fig. 2.4 is taken from the bookstore formchart in Fig. 5.10.

2.1.6 System Messages as Reports

In our definition of submit/response style systems the elements of human–computer interaction are messages from the user to the system on the one

hand, and messages from the system to the user on the other hand. We now outline the structure of these messages by giving a first account of a part of the type system for these messages.

Messages from the system to the user are conceived as reports. A report is a structured document, i.e., a document with a fixed format. A typical case would be a simple table: the document can then be seen as a list of data records, each data record being a row in the table. In the viewpoint we want to adopt in our considerations, we consider only the true data as the content of the form. This viewpoint is a little bit tricky, since usually you would require a report to contain some explanatory data, namely first and foremost the header of the table containing the column names. But in our considerations here it is of course desirable to abstract from these requirements in order to concentrate on the business logic. Now, we give one possible solution, which we will use in our following considerations. We assume that the conceptual session terminal has access to the type definitions of our system. This means that the terminal can access type definitions for rendering. This is exactly what we assume to happen with the reports the system sends to the user: the terminal knows the type of the message that is sent. Hence the browser can retrieve the column names, which are nothing more than the role names in the record type, from the user's access to the type system.

We are not interested here in how type information and business data together are rendered within a single page, as would be necessary in a typical page description language like HTML. This is not because this is uninteresting or trivial; rather our observation is that the main priority of a business application is to deliver the information in *some* readable form – as you may see in the discussions on classical form-based systems, advanced rendering topics like fonts etc. are quite neglected by these technologies. The important idea we have to remember from these considerations is the following: on our level of abstraction, the task of defining a report consists only in declaring the information content of the report as a type, and specifying with which data it should actually be filled. Of course reports can have more complex structure than the aforementioned tables. Quite common examples are groups of tables (bestsellers in a category of our bookshop, followed by subcategories), or more complex data structures within a table cell (a table with a hotel in each row, and in one column a list of pictograms for the luxury features in this hotel). In our view we suppose for each report data definition a standard rendering through the abstract browser.

2.1.7 User Input by Forms

As we have said, we see the user input as a message to the system. In our model, in close correspondence to what you will find in current platforms, the user first fills out a form in a local communication with the terminal. The state of the form in preparation is buffered within the terminal. Only at the time when the user submits the form is all the data of the form transmitted to the

system. The very notion of a form is that the data within the form adhere to a format, i.e., a data type. Hence the most important part of defining a form is to choose the message data type to which the filled forms will later belong. In our notion of the terminal this information is, moreover, the only thing this terminal needs in order to render the form. The terminal therefore basically uses the type definition itself as a key to the description of a form. Beyond that there will be a possibility to give further specifications in the single form. The bottom line is that a form in our system metaphor is a facility with which the user can construct an instance of a given data type within the terminal, and then submit it or forget it. It has to be said that such a definition of form immediately uncovers some shortcomings of many current form description languages, e.g., HTML again. That is, a straightforward type of form, which we could well need in certain circumstances, is a form where one can input a table, such as a list of records. A good example would be a form for an invoice with many single posts, where each post is mainly described by plain text. At submission you want to transmit the whole invoice to the system. Such a list, which should of course grow as you insert new lines, is a straightforward concept in our view. It is primarily absent in HTML, but it is present in some-tool based form editors from database vendors.

2.1.8 Interdependence of System Response and User Input

A key concept with regard to form-based systems deals with the way the use input can be connected with previous system responses. Forms can contain so-called selection interaction options. The form contains some kind of list, either a list of products, or a list of options or anything else. The user can choose from this list only one object – in which case it is called single selection – or an arbitrary number – in which case it is called multiple selection. The crucial point is here that the user can only choose what was offered by the system beforehand. The offer by the system can well have the status of a report in itself, e.g., in the case of the list of products. Hence the user returns to the system data which he or she had received with a report beforehand. In our model we indicate this by the fact that the report of the system contains not only readable data, but also references to entities in the system. With the selection interaction option of the form the user chooses between these references and sends them back to the system. In our notion, incidentally, this is also the only thing the user can do with these references. They are not human-readable, they are in that sense not even generally machine-readable. We will see in a later section on automated systems communications that in the same way other systems can get such references. They can only use these references in order to send them back to the system. References therefore belong to different reference types. A form field for references has such a type, and only references of this type can be returned in this form field.

2.1.9 Forms and the Bipartite Finite State Machine

The bipartite finite state machine specifies for each page type which server action types the user can submit in this page type. Therefore the forms available for the user in each page type are statically fixed. The formchart as a bipartite state machine gives a model that has an explicit notion of whether the same business functionality is available from two different page states or not. The forms on the page, however, may contain different additional specifications, which may differ from page to page. This will be a topic in the discussion of dialogue annotation in Sect. 5.4.

The best way to characterize the viewpoint which is taken by the formchart with respect to the unit system is the following. The unit system offers a set of server actions, each of which is available for the user depending on certain conditions, especially the page state. The formchart has the task of coordinating the invocation of server actions. It ensures the alternating message interchange through its bipartite structure. Furthermore it makes server actions only available for the user in the way described by the page/server transitions. In the context of interactions between systems we will discuss message communication in general.

2.2 A Message-Based Model of Data Interchange

Many enterprise systems communicate by automated interfaces. Such communication has recently become an area of increased interest in the discussions about Web services. Though Web services aim at being specifically lightweight and try to open up new applications for automated communication, the principle of automated communication is well established within technologies such as EDI. Web services are an implementation technique, not a conceptual notion. The keyword business-to-business refers partly to well-established technology, partly to new initiatives to widen the use of intersystem communication.

In our method we again want to create a homogeneous abstraction level, on which only business logic is modeled. Form-oriented analysis is only concerned with the analysis-level view of such services.

In order to achieve this unit systems can have not only human–computer interfaces, but interfaces to other unit systems as well, namely the service interfaces. In this book we will discuss modeling techniques for such service interfaces, which will have a number of commonalities with the formchart approach. In our method unit systems communicate first hand with messages. The unit system model will now especially support a transactional paradigm. This transactional paradigm offers a convenient approach to model the system behavior with respect to a message. The system response to a message can be specified as if the system can afford to process all messages sequentially, one at a time. In exchange for this it is necessary that the unit system can

communicate to other unit systems only after processing the message, by
sending new messages.

This can be put into a definition as follows. The *transactional unit system*
is a computational automaton with a state. The unit system offers a set of
transactions it can perform. Each transaction has an associated message type.
A transaction takes a message and produces a set of new messages.

An untyped view of *computational automata* is specified by a single *state
transition function*:

```
stateTransitionFunction: message × state → state × listof(message)
```

In the statically typed view the state transition function invokes for each
message type a different transaction, which is the state transition for this
message type. The transition function can provide only a fixed number of
message types:

$$\texttt{transaction}_a: \texttt{message}_a \times \texttt{state} \rightarrow \quad \texttt{state}$$
$$\times \texttt{ listof}(\texttt{message}_{a_1})$$
$$\times \texttt{ listof}(\texttt{message}_{a_2})$$
$$...$$
$$\times \texttt{ listof}(\texttt{message}_{a_{n_a}})$$

$$\texttt{transaction}_b: \texttt{message}_b \times \texttt{state} \rightarrow \quad \texttt{state}$$
$$\times \texttt{ listof}(\texttt{message}_{b_1})$$
$$\times \texttt{ listof}(\texttt{message}_{b_2})$$
$$...$$
$$\times \texttt{ listof}(\texttt{message}_{b_{n_b}})$$

$$\vdots$$

In this model, incoming messages are processed in a strictly sequential man-
ner. It may be noted here that this can be seen as one of the major services
and achievements of transactional technology to provide serial operational se-
mantics for the application programmer, while behind the scenes sophisticated
technology like locking protocols allows for a partially parallel execution. The
main task of transactional infrastructures is therefore to provide simple seman-
tics to the application programmer, which would not be scalable if translated
directly into an implementation.

Messages in our system view must have a sender and a recipient. There is
an inherent network model which takes care of the correct delivery and pro-
vision of correct sender information. The static type system is also involved
here, in that it requires the outgoing messages to be statically tagged with
the correct sender. One of the services which will be provided by the sub-
mit/response style interface view is to deal with the need for the sender and
recipient of messages in the context of a terminal session where both remain

the same during the session. For the terminal the need for a recipient is fully transparent; for the unit system the interesting data are the user ID, not the terminal.

Today there is considerable effort to specify such inter-system interfaces in which a complex protocol has to be observed. Traditional analysis techniques like structured analysis allow for no specification of such complex protocols. Form-oriented analysis offers more specification options for service interfaces through the possibility to specify the actions connected with the messages.