Chapter 4

Form Charts and Dialogue Constraints

In Chapter 2 we have seen Form-Oriented Analysis in action for a typical example system and have gained a learning-by-doing acquaintance with the concepts of Form-Oriented Analysis. The goal of this chapter in contrast is to give a terse definition of the form chart artifact, the final and most expressive artifact of Form-Oriented Analysis. The definition in this chapter is still aimed at being precise yet not formal, and in Chapter 5 we will give a formal definition of form charts in the context of the UML.

Let us recall the basic viewpoint of Form-Oriented Analysis. Form-Oriented Analysis is an approach tailored to the modeling of submit/response style applications. The form chart models the system interface as a bipartite state transition diagram and relates it to a semantic data model. The form chart can be annotated with declarative dialogue constraints based on an OCL extension. Submit/response style interaction is two staged; the interaction is divided into page interaction, which is temporary and logically local to the client until a submit is performed, and page change, i.e. a submit action. Forms and links can be conceptually unified. Only page change can affect the system data state. Hence the model is two tiered already on the analysis level. The core system state in this view does not include the client state, which is the browser's state. The advantage of this software system paradigm is that the client is well understood independent from the application. Other types of software may also use form like interfaces, but are not submit/response style, e.g. the aforementioned desktop databases as well as spreadsheet applications found in office suites. They have a single staged interaction paradigm in which each change is directly a change of the system data state. Form-Oriented Analysis abstracts from page interaction and views a page change always as a method call. In Form-Oriented Analysis strong typing is maintained at the system interface.

In this chapter a succinct description of the modeling elements of the form chart is given. Feature decomposition is explained and different degrees of com-

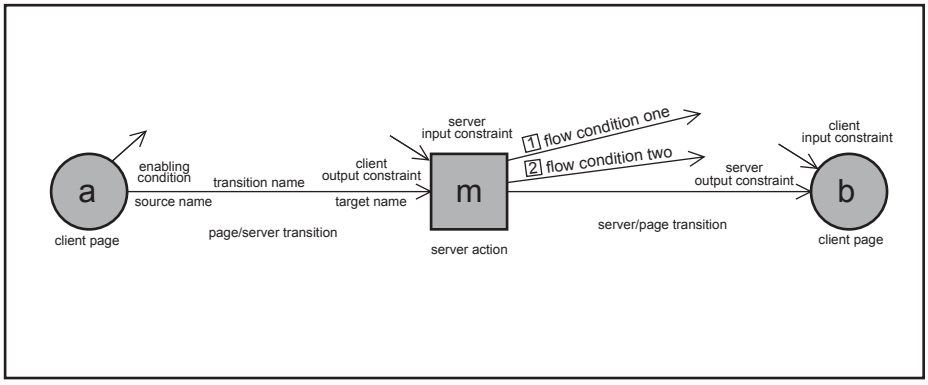

Figure 4.1: Form chart notational elements

pleteness for Form-Oriented Analysis models are defined.

4.1 The Form Chart

The user interaction with the system, called dialogue in the following, is a sequence of interchanging client states and server states. A client state presents information to the user and offers several capabilities of entering and submitting data. The client state is called client page in the following. By submitting data the dialogue changes into a server state. In the server state submitted data is processed and depending on the current core system state the generation of a new client page is triggered, i.e. the server state is left automatically. Submitting data is conceptually like calling a method, the data being an actual parameter. Therefore the server state is called server action in the following. The transition to a client page is again considered the sending of a message, this time executed automatically from the server.

Client states, server states and transitions between them form a bipartite transition diagram. The state transition diagrams used in Form-Oriented Analysis are called form charts. The form chart is annotated by declarative dialogue constraints, written in DCL, an extension of OCL.

The Dialogue Constraint Language DCL introduces special purpose constraint types, which are shown in Figure 4.1. Transitions from client pages to server actions, page/server transitions for short, host two kinds of constraints, namely enabling conditions and client output constraints. An enabling condition specifies under which circumstances this transition is enabled, based on the state during the last server action. The enabling condition may depend on the current dialogue history. The data submitted from a client page is constrained by the client output constraint. Server actions host server input constraints. They are server action preconditions in an incompletely specified system, they must be transformed to other conditions. Transitions from server actions to

client pages, called server/page transitions for short, host flow conditions and server output constraints. The flow conditions specify for each outgoing transition, under which condition it is actually chosen. The server output constraint determines which information is presented on the client page that follows in the sequel. The client input constraint is a constraint on the information on the client page, which is independent from the server page.

The constraints in the form chart are written in a variant of OCL [32]. For this purpose OCL is enriched by new contexts and key labels with appropriate semantics due to the needs of dialogue constraint writing. OCL has been chosen as a basis for the resulting dialogue constraint language despite of its lack of formal semantics [64]. Main arguments are the rich terminology introduced with OCL, its clear informal semantics and most important its usability and expressibility concerning e.g. navigations compared to other alternatives for data type annotation languages [62]. Consequently data modeling is done with the pure data kernel of UML [73], whereby we distinguish message types in the so-called data dictionary from persistent data within the semantic data model. Persistent data can be accompanied by ephemeral session related data. The system functionality is seen as side effects of server actions. It may be specified in the context of the server action, but it typically will be structured by functional decomposition.

4.1.1 States

In the form chart, client pages are depicted by bubbles, server states are depicted by rectangles. Every state, i.e. every client page and every server action must be given a name with lowercase initial. Every state has a signature, which is introduced as an OCL Type. It is defined by a UML class in the data dictionary. This defining class must have the capitalized name of the respective state and must be stereotyped as message. The message is the signature of the state as a method. Every ingoing transition of a state represents a submission capability on the preceding client page, which can be seen as a method call with the same signature. Because the signature is combined in one single parameter for every state the term superparameter is introduced for the object representing the state visit together with the signature. In accordance with the objective of writing powerful declarative dialogue constraints, superparameters must be understood as deep unchangeable. The superparameter must not be in any way mistaken as parameterized state in the sense of expressing an internal multiplicity of the state. The internal multiplicity of the state is rather given by the session data in the semantic data model.

4.1.2 Client Pages

The signature of a client page serves as abstract description of the information presented to the user. Form-Oriented Analysis does not address layout specification. Beyond the provided information a client page offers one or more data submission capabilities to the user. Every page/server transition specifies that

the respective client page has a submission capability that calls the respective server action and provides a superparameter. In a form chart a page/server-transition may be context for OCL constraints. These constraints are either client output constraints or enabling conditions, distinguished by an appropriate label. Note that a transition may be labeled with a transition name, source name and target name by the modeler. If not explicitly provided, these names are derived in an obvious way from the names of the involved states, e.g. such a transition is referred to as `a to b`. If two states are connected by more than one transition, all but one transition must be explicitly named.

A transition without client output constraint represents a data submission capability that is completely editable by the user. A client output constraint is a constraint on the actual parameters that must be ensured by the client page. Actual parameters that are constrained by a client output constraint must either not be editable and correctly provided e.g. as hidden parameters, or a client-side check must prevent data not fulfilling the constraint from being submitted. An important usage of client output constraints is to specify, that a certain actual parameter must be provided by selection from a set offered on the page. Consider the home page of our example system, which is presenting the user the participant list. The opaque references are not shown to the user, but are used as hidden parameters in the links for e.g. deletion. Clicking a link will trigger the generation of a page that presents to the user the confirmation dialogue and therefore offers further dialogue options. The client output constraint is written in the textual document attached to the form chart in the following way:

```
list to deleteLink {
clientOutput:
source.participants.person->includes(target.person)
}
```

In the above constraint the transition target name `target` refers to the actual parameter that will be transmitted to the server action. The source name `source` refers to the actual parameter of the client page. The explanation of enabling conditions is given after explaining server actions.

A client page itself may be context for a client input constraint. This constraint must hold for all client page superparameters, independent from which server action they are provided. In our example the homepage has the client input constraint specifying, that it shows all participants in ordered sequence. The singleton class PersonTable is necessary for obtaining an ordered list of participants in OCL.

```
list {
clientInput:
this.participants.name = PersonTable.participants.personData.name and
this.participants.phone = PersonTable.participants.personData.phone
}
```

4.1.3 Server Actions

A server action processes submitted data. Outgoing transitions lead to client pages. These transitions are annotated with flow conditions, which are logically mutually exclusive OCL-expressions. For one of the transitions the flow condition may be omitted, having the semantics of an "else" clause. As shorthand notation ensuring logical exclusiveness the modeler may number the outgoing transitions to enforce an evaluation order. Based on the flow conditions exactly one of the outgoing transitions is determined after server action processing. The client page that is targeted by this transition is now rendered. For this purpose the server action provides an instance of the client page data dictionary type and fulfills the server output constraint that is annotated at the relevant transition. In general there may be more than one transition between a server action and a client page, used to model conditional computation of different client page contents. A server input constraint for a server action indicates that the system is not yet completely modeled with respect to the system's behavior upon violation of this constraint. Consider the server action for changing user data. The following server input constraint would express that a password submitted by the user must be valid.

```
changeForm {
serverInput:
this.passwd=this.person.passwd
}
```

Later this server input constraint is replaced, typically by a flow condition on a transition that deals with the opposite case. The following flow condition specifies that the changed data is not accepted until the password is valid. The dialogue returns to changePage.

```
changeForm to changePage {
flow:
source.passwd<>source.person.passwd
}
```

A server input constraint can be specified as the complement of all flow conditions, like the empty flow condition. Such a server input constraint is written as a single exclamation mark. A typical pattern is the replacement of such a server input constraint by a server/page transition with empty flow condition.

A server output condition specifies the content of a client page targeted by a server/page transition. In contrast to a client input constraint this constraint is used to specify page content, which depends on the way the client page is entered. The following server output constraint specifies, that if the flow condition shown above is true, an error message is shown to the user.

```
changeForm to changePage {
serverOutput:
target.errormessage = "invalid password"
// ... further constraints  ...
}
```

As described so far the recommended server action specification already provides a tight description covering all functional aspects of this kind of system component. Furthermore nothing of the effort made in server action specification is overhead because all found constraints might be reused in system implementation. Beyond this our approach does not prescribe how to specify the data processing associated with a server action, i.e. the side effect on the system data state. Every ad hoc pseudo code notation may serve for this purpose. We recommend refraining from describing this type of functionality by any kind of artificial pre/post-condition specification that necessarily uses some modal operator. Instead we offer a constraint label sideEffect, which is used to host pseudocode describing the side effect. Annotations for side effects can be made in all contexts of Form-Oriented Analysis, on states as well as edges. A typical context is a server/page transition. Such a side effect specifies the state change under a certain flow condition. The sideEffect annotation has of course the full transition context, i.e. can access source as well as target. An unconditional side effect of a server action can be specified in the server action itself.

4.1.4 Enabling Conditions and the "along" Property

An enabling condition for a page/server transition specifies, whether a submission capability is offered to the user. In our example application a typical enabling condition would specify that the newLink is offered only, if the maximal number of participants is not yet enrolled.

```
list to newLink {
enabling:
Person.allinstances()->count() < MAX_PARTICIPANTS
}
```

Enabling conditions cannot only depend on the current system state, but they can also depend on the history of the dialogue that led to the current client dialogue state. In order to express such constraints the new OCL property "along" is introduced which can be applied to a path expression consisting of state names and describing a path in the form chart. The resulting expression evaluates to true if the current transition's source client page has been entered through states as specified in the path. This notation element makes enabling conditions a key concept for flexible and succinct modeling of even complex use

cases. Though the form-oriented approach to software engineering is not use-case driven, but feature driven, use cases as an informal notion can be considered in Form-Oriented Analysis. In the form chart every path can be considered a use case if appropriate.

In the example in Figure 4.2) a system is described by two semantically equivalent form charts. The system has two major use cases A and B. At a certain point in each of the use cases a supporting use case S may be entered which is the same in both cases. After finishing the supporting use case, the respective major use case is re-entered. The first description does not use enabling conditions. Instead it makes use of the possibility that a state may occur more than once in a form chart. We explain the semantics of this concept in the next chapter. In the alternative second description the following enabling conditions are used.

```
s5 to a5 {
enabled:  s3.s2.s1.a4->along() or s4.s2.s1.a4->along()
}

s5 to b3 {
enabled:  s3.s2.s1.b2->along() or s4.s2.s1.b2->along()
}
```

Each of the description styles has its advantages because there are tradeoffs concerning global and local complexity and ease of understanding with respect to the whole diagram and a single diagram state. A simple instance of the above example is found in web shops. The customer can enter the ordering use case in nearly every situation. After completing the ordering subdialogue the user wants to be offered an explicit link to the dialogue state from which she has once entered the ordering, i.e. she does not want to be forced to use the browser's history mechanism for this purpose.

There may be more than one transition between the same client page and server action. These transitions must carry explicit distinguishing labels. The need of several transitions is obvious with respect to enabling conditions and client output constraints.

4.1.5 Representation of Widget Types in Client Output Constraints

Form-Oriented Analysis abstracts from layout in the modeling of submit/response style systems. Therefore it is important to understand, how we can ascend from layout centric widgets to an abstract representation of interaction. In this section we analyze typical interaction patterns and discuss, how they are represented in client output constraints.

Widgets can be e.g. editable fields for primitive types, checkboxes or radio button lists. Since we deal with the analysis level, the differences between functionally equivalent widget types are of minor interest. Therefore a list

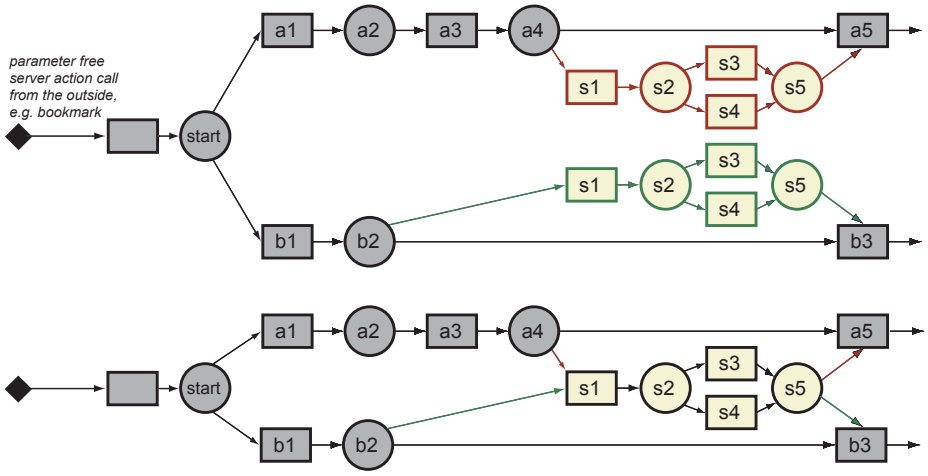

Figure 4.2: Modeling enabling conditions based on multiple state occurrences

of submit buttons is equivalent to a single submit button and either a radio button list or a selection list. Hence single selection lists, radio button lists as well as submit button lists are conceived as particular presentations of a single abstract interaction option, which we call single selection. We remember, that it was depicted in the form storyboard by an arrowhead. Checkbox lists as well as multiple select lists we consider accordingly as multiple selection. In both selection types, single selection as well as multiple selection, the user can chose from an offered collection, which must be part of the client page signature. In case the user has to choose between a set of primitive values, these values are wrapped as objects. This technique resembles the flyweight pattern [6] from the design phase.

Selection Widgets

Selections in form charts give rise to a constraint between the offered collection in the client page signature and the selected items that are part of the addressed server action signature. In that server action signature the parameter that has to be provided by the radio button list must be part of the offered collection in the client page signature from which to choose. This constraint is a client output constraint in Form-Oriented Analysis. Such a client output constraint written in OCL was discussed above when it was used in our example application. It is graphically shown in Figure 4.3, but for simplicity we assume there a data dictionary slightly different from our running example, namely we assume that the collection is top level part of the client page signature.

For single selections the association representing the chosen object has accordingly the cardinality 1. For multiple selections the association has unspecified cardinality. The condition that each element of the offered collection is

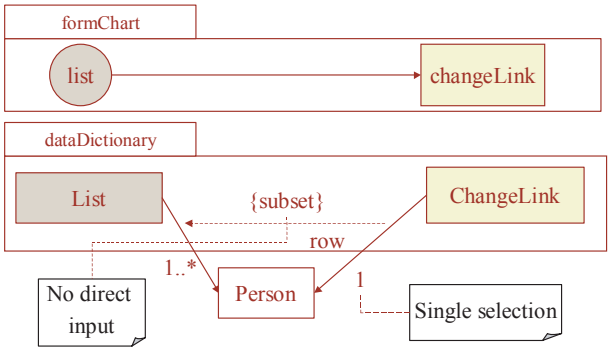

Figure 4.3: Graphical representation of a client output constraint. The subset constraint specifies that the chosen information must have been on the page.

selected only once is a direct consequence of the fact that associations are set valued. These considerations hint immediately on a further possible type of selection widget which would enable bag relational selection by using the *ordered* annotation for the chosen collection.

Direct Input

The counterpart to selection is given by direct input fields. Direct input fields are fully editable, but may be prefilled with a default parameter. Direct input fields are applicable only to primitive types. Direct input fields may, but are not required to have client output constraints, e.g. pure string input may have no constraints. A typical nontrivial client output constraint is a constraint that a time or date must lie in the future.

```
somePage to someAction {
clientOutput:
this.date > System.time
}
```

4.2 Feature Composition

Feature composition has been introduced as the composition mechanism for form storyboards as well as form charts. The graph structure of a form storyboard or a form chart has been specified as being a bipartite directed labeled multigraph. Every sub graph of the form chart is called a feature chart. Two feature charts are combined by graph union. A form chart decomposition is a collection of feature charts in such a way that the combination of the feature charts yields the complete form chart.

The perhaps most intuitive explanation, why feature composition is possible and meaningful in Form-Oriented Analysis is the inverse operation, feature

decomposition. A complete diagram of Form-Oriented Analysis, be it a page diagram, a form storyboard or a form chart, has a uniquely stable semantics: If page/server edges, i.e. interaction options are removed, the data integrity is not endangered. Certain usages of the system may of course become impossible, if one removes key interaction options for the system. But the semantic data model is not corrupted by such operations: the system remains stable, if it was stable before. As a consequence the form chart covers system behavior that is inherently stable against runtime customizations.

The composition of the analysis model is of course especially important with respect to the task of expressing preferences and priorities in the system specification, as well as to enable the discussion of alternatives and trade-offs between them.

4.2.1 Compatibility Issues

There are some rules for the composition of two features. The rules follow from the fact that the features to merge must be subgraphs of one single form chart. First no node is at the same time client page in one graph and server action in the other. Nodes of the same name must have the same data dictionary type, because different features are different form charts over the same data dictionary and model.

If two features are combined, the constraints have to be compatible. If in a feature composition step a server action receives server/page transitions from different features, the flow condition numbers in both features must be different in order to be merged into a single order unless they are mutually exclusive. The server/page transition without flow condition has to be the same in both features, or one of the features should have no server/page transition without flow condition.

4.2.2 Hierarchical Feature Decomposition

A form chart can be decomposed in a hierarchical manner. The result is a tree of chart decompositions. Decomposition makes the form chart manageable. It is a tool for organizing the form chart artifact during the analysis phase. The feature hierarchy as such is not semantically relevant for the specification. Every combination of feature charts, even from different levels of the tree, yields a correct sub graph of the form chart.

4.2.3 Menu-like User Interface Parts

An important special case of feature composition is the modeling of menu-like options, i.e. interaction options, which are offered on many, perhaps even all pages. A new notation element for this purpose is the state set, that is depicted by a double lined state icon. It is annotated by a list of state names and serves as shorthand notation for these states. The example in Figure 4.4 shows page sets. An edge between two state sets of say m client pages and n server

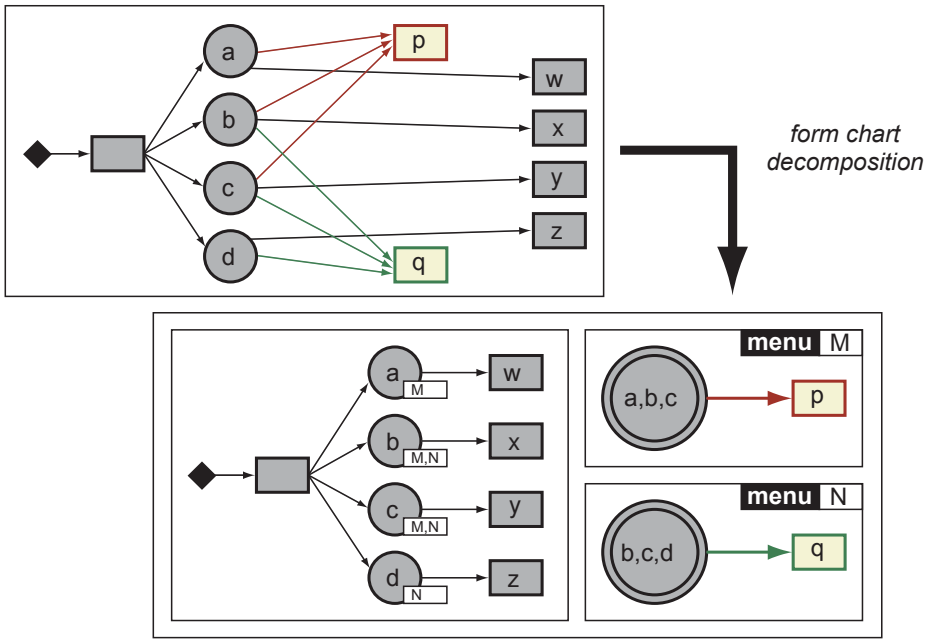

Figure 4.4: Modeling menu-like user interface parts

actions represents the complete bipartite graph $K_{m,n}$ between the elements of the state sets. A feature chart may be annotated as menu. Then the page/server transitions contained in this feature must not be contained in the main form chart or its decompositions. Affected states may reference the respective menu feature chart by an explicitly given name. Figure 4.4 shows how the described mechanism fosters readability of system interfaces with menu-like user interface parts. Another notation flavor is to give the state set a single name, and to reference the page set in its member states. The menu construct is used in the form chart of the seminar registration system in order to model the home button.

4.2.4 Refinement

Form-Oriented Analysis allows for different grades of detail. It does not require complete specification. We do not confine Form-Oriented Analysis to a single process model in this paper. Instead we provide a set of well-defined incomplete abstraction layers called refinement stages. Refinement itself is defined by using the notion of feature: the model B is a refinement of model A iff A is a feature of B. In an informal sense form charts can be seen as refinements of form storyboards. We now explain the most important refinement stages:

Signature Model

This model contains the complete data model and the complete form chart, but no constraint annotations in both diagrams. This model is valuable as the bare metal model giving the complete structure of the user interface and the data.

Server Input Declared Model and Server Input Safe Model

Server input constraints have been explained as being related to user input that does not meet the requirements, e.g. the user enters a sum above its limit. Server input constraints have to be replaced in later stages by branches from the server page for these cases. These branches are of minor interest, therefore it is helpful if their full specification can be deferred. A server input declared model is the model that contains server input constraints. The server input safe model is the model where all server input constraints are replaced by branches leaving the server state.

Multi-User Declared Model and Multi-User Safe Model

Submit/response style applications are often multi-user systems. Therefore during the usage one has to consider many clients enacting on the same semantic data model. Each client has its own instance of the finite state machine. These instances are completely independent and interact only via the semantic data model. Single executions of the server action of different instances of the form chart are considered as mutually exclusive and not interfering with each other, i.e. the semantic data model is seen as accessed in a virtually serial manner. This is a helpful viewpoint for the specification phase, and is fully compatible with later development stages of typical form based applications. However, only the single server action is executed atomic in this sense. Hence subsequent interaction of the same user with the system can be influenced by the interaction of other users. Such effects are therefore called multi-user exceptions. These exceptions are due to the fact that submit/response style systems are typically based on an optimistic business logic approach. Well-known examples are systems with shopping carts. A typical strategy is that the items in the shopping cart are not reserved for the customer. It is assumed a rare event that the item has been sold by the time the customer finally buys the content of the whole shopping cart. This assumption exactly is the optimistic assumption, and vice versa the multi-user exception occurs whenever the optimistic assumption fails. In our example system the dialogue for enrollment follows an optimistic strategy. The enrollment link is offered, as long as free places in the course are available. The event considered rare in this context is the case that a student starts the enrollment, and when he has filled out his form, he finds all places occupied. For many purposes the modeler may want to use a model that abstracts from such multi-user problems, and therefore does not contain all possible exceptions of this type. Such a model, which will be the primary working model, we call the multi-user abstracted model. The multi-user declared model is the model in

which all multi-user exceptions are excluded by server input constraints. The multi-user safe model is the model in which these constraints are replaced by branches leaving the server state.

4.3 Advanced Topics

4.3.1 Active Content

Form chart allows for certain system caused page updates during a single client page. A typical example is the inbox in a mail system. The inbox page shows the list of arrived mails. A desired behavior is, that incoming mails are immediately shown to the user by appending them to the list. Pages, which offer such a server cause update we call pages with active content. Form-Oriented Analysis allows only for the augmentation of the current data dictionary object, which is shown. The inbox is an example of a list, which receives additional elements. Another example is the disabling of single selection options in a list. The active content does not violate the overall system metaphor of Form-Oriented Analysis, that page change is triggered by the user. The active content is conceptually on the level of the page interaction of the user. Page changes cannot be triggered by the system.

Active content must be explicitly indicated by an ampersand at the enabling condition. Active content is typically necessary only at a small number of places. A typical system model would put all active content in separate features. Sometimes a menu feature has an active component, specifying a single flag, which is visible on all pages, e.g. a flag indicating new mail.

4.3.2 Communication with Other Systems

Frequently systems communicate by automated interfaces. Such communication has become recently an area of increased interest through the discussion 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. The keyword business-to-business hence refers partly to well-established technology, partly to new initiatives to widen the use of inter-system communication. Form-Oriented Analysis is only concerned with the analysis-level view on such services. In Form-Oriented Analysis such interfaces between systems are called external interfaces.

The key argument, why external interfaces are semantically unproblematic in Form-Oriented Analysis is that ingoing messages can never change the state of the form chart directly. The ingoing messages only changes the system state i.e. the state of the semantic data model directly. This principle is a direct consequence of the fact that form charts represent pull style interfaces.

External interfaces in Form-Oriented Analysis are completely message based i.e. they are either ports for ingoing messages or ports for outgoing messages.

The specification of external interfaces is straightforward in Form-Oriented Analysis. Specifically outgoing interfaces are just part of the side effect of server actions. Each server action can send an arbitrary number of messages. Ingoing interfaces on the other hand require an own class of updates. Such updates have one ingoing message type from the data dictionary, they can specify an update on the semantic data model and can again send an arbitrary number of messages. Ingoing external interfaces are global update operations on the semantic data model.

Today there is intensive effort to specify such inter-system interfaces in which a complex protocol has to be observed. Traditional analysis techniques like Structured Analysis allows for no specification of such complex protocols. Form-Oriented Analysis offers more specification options through the use of OCL constraints in accordance to standard object-oriented specification methods.