Appendix A

Example Dialogue Constraints

The following listing shows all dialogue constraints of the seminar registration system. These constraints are given in the attachment to the form chart. We give after each constraint a comment to the meaning of the constraint.

The model we discuss in the following is a multi-user abstracted model, i.e. it relates to an optimistic viewpoint of multiuser exceptions. In Section A.1 we discuss the muti-user safe model.

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

This constraint specifies, which information is shown on the home page of the system. This information is independent of the transition, over which the page is entered, therefore it is appropriate to use a client input constraint. The right hand side expressions have an ordered list as a result. Therefore the left hand side expressions must specify the same ordered list, too. By this way, it is implicitly specified, that the page contains one record for each participant, and that they are ordered in the same way as in the sematic data model.

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

This edge represents an interaction option, which is a selection. The constraint specifies, that the chosen entry must be in the list shown on the home page.

The cardinality, i.e. that it is a single selection is already given by the type specification of ChangeLink.

```
changeLink to changePage {
serverOutput:
target.errorMsg = ""
target.person = source.person
target.personData->equals(source.person.personData)
}
```

The comparsion expression `equals()` returns true, iff the compared objects `target.personData` and `source.person.personData` are clones, i.e. have the same attribute values.

If changePage is entered from the list page, the current data record of the user is presented as default values for the input fields.

For the changePage the information shown on the page is dependent on the transition the page is entered through, therefore the content of the page is specified on each server/page transition leading to the changePage as a server output condition. This constraint should be compared with the server oputput constraint on the changeForm to changePage transition coming later.

```
changePage to changeForm {
clientOutput:
source.person = target.person
}
```

The input elements of the changeForm are direct input fields, therefore they do not impose any client output constraints. The only client output constraint deals with the hidden parameter, which is an opaque reference.

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

The case of an invalid password.

```
changeForm to changePage {
serverOutput:
target.errormessage = "invalid password"
target.personData->equals(source.personData)
}
```

In case of an invalid password the edited field values are shown as new default values. The page content specification differs not only in the error message from the changeLink to changePage server output constraint.

```
changeForm to list {
sideEffect:
update
source.person.personData->equals(source.personData)
}
```

In case of correct authentication the new user data are inserted into the persistent data.

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

This constraint corresponds to the list to changeLink constraint, it specifies a single selection.

```
deleteLink to deletePage {
serverOutput:
target.errorMsg = ""
target.person = source.person
target.name = source.person.personData.name
}
```

The page displays the user name.

```
deletePage to deleteForm {
clientOutput:
source.person = target.person
}
```

The constraint marks the passing of the opaque reference of the record under consideration. The passing of the password is again implicitly specified by the type definition of DeleteForm.

```
deleteForm to deletePage {
flow:
source.passwd<>source.person.passwd
}
```

This is the flow condition for the case that the password is invalid. The dialogue returns to the deletePage.

```
deleteForm to deletePage {
serverOutput:
target.errormessage = "invalid password"
target.person = source.person
target.name = source.person.personData.name
}
```

For the case of an invalid password, the deletePage is reentered, with an error message.

```
deleteForm to list {
sideEffect:
delete source.person
}
```

The deletion of a record shall only be performed, if the password was valid, hence the sideeffect is annotated to the corresponding server/page transition.

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

This condition says, that the link to the new registrations is only accessible, as long as the number of participants is below the threshold. In this example the threshold is intended to be a soft threshold in the following sense. If accidentally several users enter the registration dialogue while only a single place is free, they shall still all be able to complete the dialogue, even if then there are more registered students. We assume that the domain experts prefer such an overbooking as preferrable to the case, that the users are disgruntled by being thrown out of a running registratio dialogue.

```
newLink to newPage {
serverOutput:
target.errorMsg = ""
target.personData.* = ""
}
```

If the page is newly entered, the fields are empty.

```
newForm to newPage {
flow:
source.passwd<>source.passwd2
}
```

The user has to retype his password choice correctly, before the data are accepted.

```
newForm to newPage {
serverOutput:
target.errormessage = "incorrectly retyped password"
target.personData->equals(source.personData)
}
```

The case, that the user has not retyped the password correctly.

```
newForm to list{
sideEffect:
insert person x
x.personData->equals(source.personData)
x.passwd = source.passwd
}
```

In this case the user input is shown as default value in the fields of the page.

A.1 Multi User Aspects

In this section we consider aspects of the system specification dealing with concurrency problems due to the use by several users. The execution of each server action is atomic. But the link-page-form pattern is the simplest example of a multistep usecase, in which the execution of the different usecases does not form a single atomic execution unit. We have assumed an optimistic viewpoint in the last section. Let us discuss the deletion subdialogue as an example. From the business logic side we expect only one client to attempt a deletion at the same time. However, if two concurrent deletion attempts are started, the system should not run into an undefined state.

In the following we list all constraints which together make the system multi-user safe. The multi-user safe model requires typically new transitions for occurring multi-user exceptions. In our example each server action gets a new outgoing transition. We direct all these transitions to the homepage, i.e. to list. All exception transitions are named, their name is the respective server action name with a suffix `...failed`.

In the multi-user safe model the threshold MAXPARTICIPANTS for the number of participants in the semantic data model is sharp.

```
newLinkFailed {
flow:
Person.allinstances()->count() < MAXPARTICIPANTS
}
```

This flow condition is true, if the maximal number of participants was reached since the time the last page was rendered. The dialogue returns to the homepage. Note that in this case the enabling condition for list to newPage is automatically false.

```
newFormFailed {
flow:
Person.allinstances()->count() < MAXPARTICIPANTS
}
```

This is the case that the user finds no free place after he has filled out the form. The dialogue returns to the homepage. Again in this case the enabling condition for list to newPage is automatically false.

```
deleteLinkFailed {
flow:
source.person->isEmpty
}
```

This flow condition is true, if the participant was deleted since the time the last page was rendered. The dialogue returns to the homepage.

```
deleteFormFailed {
flow:
source.person->isEmpty
}
```

This flow condition is true, if the participant was deleted since the time the last page was rendered. The dialogue returns to the homepage.

```
changeLinkFailed {
flow:
source.person->isEmpty
}
```

Fully analogous to deleteLinkfailed.

```
changeFormFailed {
flow:
source.person->isEmpty
}
```

Fully analogous to deleteFormFailed.

The option to enter the delete dialogue from the change page does not require an additional multi-user constraint. In the case the user follows this option, the condition for deleteLinkfailed is guaranteeing multi-user safety.