Chapter 6

Telephone example

§6 Telephone example

In this chapter we write an LSD specification to model the behaviour of a simple telephone system. We then consider the issues of synchronisation and interaction between agents which arise from the specification. Using the techniques of chapter 5 we show how to transform the LSD specification into a definitive program for the adm. The parameters of the transformation are identified, so that the transformation gives a family of programs. Appendix 7 contains examples based on this chapter.

The purpose of this chapter is to illustrate how our method for transforming a model of a system described by an LSD specification into an executable program allows simulaton of the behaviour of the system.

§6.1 Telephone specification

In Figure 6.1 we give a specification for a simple telephone system. The telephone specification is based on one given in [Beynon 86]. The use of parameters makes the specification extensible to a multi-user, multi-telephone system. We consider a user `U` and a source telephone `S`. The user may attempt to telephone a destination telephone `D`. Other users may connect to telephone `S`.

When a user picks up the receiver of a telephone which is not ringing a `dial()` agent is instantiated. If the user dials a number a `connect()` agent is instantiated by the `dial()` agent, and the `dial()` agent is deleted. The `connect()` agent is extant for the duration of the call. Once the connection is made `connected` becomes true and the telephone starts ringing. If the line is engaged, the `connect()` agent is deleted.

```
agent user(U,S) {
oracle
        (int) tone[S], (bool) ringing[S]
state   (bool) onhook[S], (int) dialled_number[S]
protocol
        ¬onhook[S] § onhook[S]= true ; dialled_number[S]=@,
        ¬onhook[S] € (tone[S] == D) § dialled_number[S]=N,
        ¬onhook[S] € (tone[S] == C) § <speak>.
        onhook[S] € ¬ringing[S] § onhook[S] = false ; dial(S),
        onhook[S] € ringing[S] § onhook[S] = false ; <speak>
}
```

```
agent telephone(S) {
oracle  (bool) #onhook[S] = true,
        (int)  #dialled_number[S],
        (bool) connected[S,*],
        (bool) #connecting[S,*] = false,
        (bool) engaged[S,*],
        (bool) #dialling[S] = false
derive
        (char) #tone[S] =
        D if dialling[S]:
        E if connecting[S,dialled_number[S]]
                                        € engaged[S,dialled_number[S]]:
        R if connecting[S,dialled_number[S]]
                                        € ¬engaged[S,dialled_number[S]]:
        C if connected[S,?]⁄connected[?,S]:
        @ otherwise
}
agent exchange() {
oracle  (bool) onhook[*], #connected[*,*] = false, connecting[*,*],
                #answered[*] = false
derive
        (bool) #ringing[*] = connected[?,*] € onhook[*]
                                                        € ¬answered[*],
        (bool) #engaged[S,*] = connecting[S,*]
                                          € (ringing[*]⁄¬onhook[*]),
        (time) #Tdial = <timeout for dialling>,
        (time) #Tcall = <timeout for calling>
}

agent dial(S) {
oracle
      (int)  dialled_number[S],
      (time) Tdial, time,
      (bool) onhook[S],
      (time) connecting[S,*] = false
state
      (time) #tstart = |time|
derive
      (bool) #valid = <dialled_number[S] is valid>,
      (bool) dialling[S] = ¬onhook[S] € ((time - tstart) < Tdial),
      (bool) LIVE = dialling[S] € ¬connecting[S,dialled_number[S]]
protocol
      dialling[S] € valid §    connect(S,dialled_number[S])

}
```

```
agent connect(S,D) {
oracle  (bool) onhook[S], onhook[D], ringing[D], engaged[S,D],
        (time) Tcall, time,
state
        (time) #tcall = |time|,
        (bool) connected[S,D] = false,
        (bool) answered[D] = false
derive
        (bool) connecting[S,D] = ¬connected[S,D],
        (bool) LIVE = ¬onhook[S] €
                      ((connecting[S,D] € (time-tcall)<Tcall))
                      ringing[D]
                      (connected[S,D] € (¬answered[D] ¬onhook[D]))
protocol
        ¬engaged[S,D] € ¬connected[S,D] § connected[S,D] = true,
        ¬onhook[D] € ¬answered[S,D] € connected[S,D]
                                          § answered[D] = true,
        engaged[S,D] § delete connect(S,D)
}


agent environment() {
state #time = 0
protocol
        true § time = |time| + 1
}
```

Figure 6.1

§6.2 Synchronisation issues

In this section we consider the synchronisation issues which arise from the telephone specification of Figure 6.1, and the intended behaviour of a simulation of the LSD specification.

Considering first the `user()` agent, we have only specified when an action may occur, not when we would like it to occur. For example, the normal behaviour of the user would be to put the receiver down after speaking, rather than immediately after picking the receiver up. We might extend the model by, for example, specifying that the average length of a telephone call which is invoked by user U is twelve minutes. In an LSD specification we can only express such a characteristic of protocol execution in a convoluted and counter-intuitive manner, for example by explicitly referring to the time that a call was started in the precondition for replacing the receiver. LSD can describe in a natural way permissions for action, but not when the action is most appropriate.

The telephone is an entirely passive unit which is possessed of certain characteristics. In other words, the `telephone()` agent does not contain a protocol, but only the declaration of variables which are owned by the telephone. Notice that some variables in the case statement defining `tone[S]` will only occasionally be evaluable,

when their owning agent is extant. The semantics of the case statement is that no more than one of the specified expressions (i.e. excluding the default case) may hold, and a non-evaluable expression is treated as not holding.

Two different uses are made of derivates within the specification (as described in §8.5). The first use is exemplified by the variable `valid`, which serves as a macro to permit a convenient shorthand within the specification of the `dial(S)` agent. The second use is shown by the definition of the derivate `dialling[S]` in `dial(S)`, which requires synchronisation between the value of `onhook[S]` as defined and changed by the `user()` agent and its value as perceived by the `dial(S)` agent.

Variables are used in the `dial()` agent to impose a maximum permissible time for dialling, after which a timeout is triggered. This means that it is possible for the `dial()` agent to terminate without performing any actions. In such a case the next action by the dialling user must be to replace the receiver. The fact that these operations must occur in a sequential manner cannot be described explicitly, because the specification is agent-oriented, so it can only be discerned implicitly by examination.

The termination of the `dial()` agent occurs when one of three conditions pertains: the receiver is replaced, the time taken to dial exceeds that permitted, or the call is valid. In the last case, the `connect()` agent is invoked, which causes the variable `connecting[S,D]` to be defined. The evaluation of the definition for `connecting[S,D]` will initially be true, and so the LIVE variable in the `dial()` agent will become false, thus terminating the agent. This technique requires synchronisation by assumption between the `dial()` and `connect()` agents. We make the assumption that the nature of the state-oracle coupling for the `connecting[S,D]` variable is such that LIVE will be set to true (and thus terminate the agent) before further instances of the `connect()` agent can be invoked from within `dial()`. In this instance we would wish to ensure that the value of `connecting[S,D]`, as viewed by the `connect()` and `dial()` agents, was updated more frequently than the protocol of the `dial()` agent was executed. To do this requires that we have some means of expressing the relative speeds of execution of agent instantiation and execution.

§6.3 Transformation of the telephone specification

In this section we present the result of transforming the LSD specification of Figure 6.1 into an adm program. We

identify the parameters of the transformation. The changing of these parameters results in different definitive programs. The program presented in §6.3.1 to §6.3.5 is listed in full in Appendix 7.1. Examples of the execution of a selection of these programs is presented in Appendix 7.2 to 7.8. An example of the interactive use of the program is given in Appendix 7.9.

Three issues are resolved by parametrising the transformation: the relative speed of execution of the agents, the nature of value propagation, and the natue of guard selection. Any variable which is not updated at every opportunity is an example of a loose synchronisation between agents, e.g. `dialled_number[_S,3]` in the `dial()` entity.

In the following sections we comment on the transformation of each agent into an entity.

§6.3.1 The `user()` entity

In this section we comment on the entity which was produced by transforming the `user()` agent in the LSD specification. No unowned variables are initialised by the `user()` agent, so there is no need to initialise unowned variables in the entity using the `init_flag` construction. The variable `only_one[_U,1]` is initialised to 1 in the definition section so that no guards from the LSD specification will be true in the first execution cycle. It is not initialised to be `|rand(50)|` because of the problem with initialisations involving evaluation which is described in the user documentation in Appendix 3.

In this program the telephone number which the user will dial is passed as a parameter to the entity, i.e. `_D`. This means that each user can only dial one telephone number. For a genuine simulation of a complete telephone system this restriction would need to be removed, and the value of `dialled_number[_S]` allowed to vary.

The guarded command

```
even && ringing[_S,1] print (_S," is ringing") -> ,
```

contains no commands in the command list - its purpose is just to invoke a procedural action in the appropriate state

to print a message that the telephone is perceived to be ringing.

The guard

```
even && (slow[_U,1] > 0) -> slow[_U,1] = |slow[_U,1]| - 1,
```

implements the control mechanism to slow the user entity down when it has dialled a number. The effect of not using this construct in the transformation process is shown in Appendix 7.6.

When one of the five guards which come from the LSD specification is evaluated as being true the variables `selected[_U,1]` and `level[_U,1]` are used to indicate how much of the guarded command has been executed. At each stage an arbitrary delay can be introduce by appending a conjunct of the form "`&& (rand(`**n**`) == 1)`" to the guard. The guard of the command to replace the receiver is

```
even && (selected[_U,1] == 1) && (level[_U,1] == 1)
                                          && (rand(3) == 1)
print ("replacing the receiver of ",_S)
                                -> onhook[_S]=true ; level[_U,1] = 2,
```

This indicates that there will be some arbitrary, but relatively short, delay between deciding to put the telephone back on the hook and actually doing so. The value chosen for **n** is 6 before dialling a number, which indicates that there is on average a longer delay between deciding to dial and actually dialling than between deciding to replace and replacing the receiver. Where there is no such conjunct this is because the value chosen for **n** was 1, and so the conjunct, which is always true, has been omitted. For example:

```
even && (selected[_U,1] == 4) && (level[_U,1] == 2)
                                -> dial(_S,_D) ; level[_U,1] = 0 ;
                                selected[_U,1] = 0,
```

which means that when the receiver has been lifted up the `dial()` entity is instantiated as soon as possible.

The act of speaking into a receiver is indicated in the simulation by a procedural action, but is a null action as far as the execution of the entity is concerned:

```
even && (selected[_U,1] == 3) && (level[_U,1] == 1)
                                          && (rand(6) == 1)
print ("speaking into ",_S)
                                -> level[_U,1] = 0 ; selected[_U,1] = 0,
```

The change of state when a user speaks is indicated by changing the values of the variables `selected[_U,1]` and `level[_U,1]`.

A cognitive argument is used to indicate that all variables should be kept as authentic as possible, which leads to the guards:

```
!even -> tone[_S,1] = |tone[_S]|,
!even -> ringing[_S,1] = |ringing[_S]|,
!even -> onhook[_S,1] = |onhook[_S]|,
!even -> dialled_number[_S,1] = |dialled_number[_S]|,
```

For example, whenever the tone of the phone changes, it is in general known immediately by the listening user. The user's perception of whether the telephone is on the hook or not is normally accurate. Similar arguments can be applied for all the variables perceived by the user entity.

§6.3.2 The `telephone()` entity

In this section we comment on the entity which was produced by transforming the `telephone()` agent in the LSD specification. No unowned variables are initialised by the `telephone()` agent. Since it does not own the `connected` or `engaged` variables, it must update the perceived values of these variables. This means that there will be the guarded commands

```
!even ->connected[_S,_D,2]=|connected[_S,_D]|,
!even -> engaged[_S,_D,2] = |engaged[_S,_D]|
```

associated with the `telephone()` entity, even though there are no guarded commands in the specification of the `telephone()` agent.

§6.3.3 The `dial()` entity

In this section we comment on the entity which was produced by transforming the `dial()` agent in the LSD

specification. This entity is instantiated by the `user()` entity when a call is to be made, and is deleted when the connection has been initiated, the calling telephone has been replaced on the hook, or too long has been taken in dialling. The `LIVE` derivate variable of the specification has been transformed by testing it whenever `even` is false

```
!even && !init_flag[_S,3] && !(dialling[_S,3]
                                    && !connecting[_S,_D,3])
print ("terminating dialler from ",_S," to ",_D)
                          -> dialling[_S] = false ;
                                 delete dial(_S,_D),
```

This ensures that updates to variables can only occur when the entity is not about to be deleted (thus avoiding a redefinition and deletion of the owning entity in the same run set). The "`dialling[_S] = false`" command resets the definition associated with `dialling[_S]` to that in which the `dial()` agent was invoked.

When it has invoked the `connect()` entity the operation of the `dial()` entity is slowed down by using the `slow[__S,3]` construct:

```
even && (level[_S,3] == 1)
print ("starting connection from ",_S," to ",_D)
                                 -> connect(_S,_D) ; level[_S,3] = 0 ;
                                 slow[_S,3] = 20,
```

This allows time for the `connect()` entity to perform the initialisations which make `connecting` true, and for this change to be perceived by the `dial()` entity.

§6.3.4 The `connect()` entity

In this section we comment on the entity which was produced by transforming the `connect()` agent in the LSD specification. The variable `tcall[_S]` is initialised in the first guarded command because of the problem with initialisations involving evaluation, which is described in the user documentation in Appendix 3.

```
init_flag[_S,4]                  -> answered[_D] = false ;
                                 tcall[_S] = |time| ;
                                 connected[_S,_D] = false ;
                                 connecting[_S,_D] = !connected[_S,_D] ;
                                 init_flag[_S,4] = false,
```

Were this not done, it would always have the definition `|time|`, i.e. the current time.

When `even` is false the value of `only_one[_S,_D,4]` is set (by the last three guarded commands in the `connect()` entity) to be the number of whichever of the guards in the LSD specification is true. If no guard is true, then the value of `only_one[_S,_D,4]` is left unchanged. This means that as soon as the connection can be made, the line is engaged, or the picking up of the ringing receiver is registered, the associated command will be selected for execution.

§6.3.5 The `exchange()` and `environment()` entities

In this section we comment on the entities which were produced by transforming the `exchange()` and `environment()` agents in the LSD specification. The use of the "*" and "?" constructs in the LSD exchange specification needs to be explicitly enumerated in the `exchange()` entity, because there are no pattern matching facilities available in `am`, for example:

```
ringing[7124] = connected[6489,7124] && onhook[7124]
                                              && !answered[7124]
```

All such variables must be explicitly declared as owned and stored in the definition store D. In the `environment()` entity there is no need to use an `only_one` variable because there is only one guarded command, which becomes:

```
even && (level[6] == 0) -> level[6] = 1,
even && (level[6] == 1) -> time = |time| + 1 ; level[6] = 0,
```

The toggling of the value of `even` occurs in the `environment()` entity, because it is an always instantiated exactly once:

```
true -> even = !|even|
```

§6.3.6 Instantiations at compile time

Once entered the definitive program is instantiated as follows:

```
user(10,6489,7124)
user(11,7124,6489)
telephone(6489,7124)
telephone(7124,6489)
exchange()
environment()
```

This causes the instantiation of two users and their associated telephones, and an exchange and environment entity. The simulation is started by the command:

```
start
```

Examples of this program and others which have used different transformation parameters are given in Appendix 7.2.