PROTOCOL SYNTHESIS 10

10.1 INTRODUCTION

One of the toughest open problems in protocol design is finding a discipline of programming that can guarantee *a priori* the derivation of a functionally correct protocol that is free of dynamic errors such as deadlock. A proper design discipline will lead to a smaller and more effective product that is easier to maintain and modify. As yet, little progress has been made in this area. This chapter is therefore necessarily tentative.

We briefly discuss three methods for interactively building correct protocol specifications. The first two of these methods focus on the functionality of a protocol design; the third emphasizes structure.

Bear in mind that a protocol synthesis method cannot synthesize service specifications. No automated tool can determine the purpose of a new protocol. The design problem is to find a protocol that (1) realizes a *given* service, and (2) does so in an error-free manner. All three methods discussed below assume that a service specification exists, either in a formalized form or informally in the mind of the user of the synthesis tool.

In the next section we illustrate a protocol derivation method that allows us to synthesize the protocol components from a formal specification. We do this by formalizing the service specification in such a way that a skeleton structure for the protocol procedure rules of each communicating process can be extracted from it. The synthesized processes can then be fine-tuned manually.

10.2 PROTOCOL DERIVATION

In protocol validation we may want verify assertions that the user makes about the structure of possible dialogues between processes. A dialogue is a sequence of message exchanges that can be observed at a given interface, e.g., a set of channels.

Consider the problem of designing a connection management protocol.

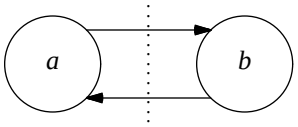

Figure 10.1 — Interface

There are two processes, *a* and *b*, that share access to a full-duplex data link, indicated with two arrows in Figure 10.1. Processes *a* and *b* have to coordinate the beginning and the ending of data transfers across the link.

Typically, the designer is asked to supply two process specifications, one for each side of the connection, in an attempt to constrain the possible dialogues to a well-defined set. An assertion about these constraints can be formalized and verified by an automated protocol validator. In protocol synthesis we can try to turn this problem around by beginning with a specification of the set of allowable dialogues and deriving the processes from them so that, by construction, these processes will be unable to exhibit any other than the stated behavior.

We provide a specification for two processes. Either side can initiate a connection; if both processes try to do so at the same time the attempt fails. The behavior can be specified as a six-state machine, as follows, in an informal notation resembling PROMELA. The notation `a->b` informally encodes the direction in which a message flows.

```
spec manager
{
idle:
        if
        :: b->a!connect -> goto b_opens
        :: a->b!connect -> goto a_opens
        fi;
a_opens:
        if
        :: b->a!accept -> goto connected
        :: b->a!connect -> goto idle    /* conflict */
        fi;
b_opens:
        if
        :: a->b!accept -> goto connected
        :: a->b!connect -> goto idle    /* conflict */
        fi;
connected:
        if
        :: b->a!disconnect -> goto b_closes
        :: a->b!disconnect -> goto a_closes
        fi;
```

```
a_closes:
        b->a!disconnect -> goto idle;
b_closes:
        a->b!disconnect -> goto idle
}
```

This specification describes the message exchanges that are visible at the interface between *a* and *b*, i.e., at the dotted line in Figure 10.1. There may be other messages that are handled by *a* or *b*, and there may be many other tests and data manipulations to be performed. The above specification is therefore partial.

Some of the messages are to be sent by process *a* and some are to be received by *a*. We can derive a skeleton process description from the specification that describes precisely the constraints for process *a*. Mechanically, we can then derive the following state machine for process *a*. We can say it is the derivative of specification *manager* with respect to *a*.

```
proctype D_manager_D_a()
{
R0:     if
        :: b!connect -> goto R1
        :: a?connect -> goto R2
        fi;
R1:     if
        :: a?connect -> goto R0
        :: a?accept -> goto R3
        fi;
R2:     if
        :: b!connect -> goto R0
        :: b!accept -> goto R3
        fi;
R3:     if
        :: b!disconnect -> goto R4
        :: a?disconnect -> goto R5
        fi;
R4:     a?disconnect -> goto R0;
R5:     b!disconnect -> goto R0
}
```

The state machine for process *b* is similar, since the protocol specification is symmetric. The derivation is trivial in this case and can easily done by hand. In general, though, the derivation is more subtle.

Consider the following example that describes the behavior of a simple alternating bit protocol. The interface is the same as shown in Figure 10.1. The specification for the messages that cross the interface at the dotted line, however, is now formalized as follows.

```
spec abp
{       do
        :: a->b!msg0;
```

```
                do
                :: b->a!ack0; break
                :: b->a!ack1; a->b!msg0
                od;
           a->b!msg1;
                do
                :: b->a!ack0; a->b!msg1
                :: b->a!ack1; break
                od
        od
}
```

This single specification completely describes the behavior of two protocol machines, the sender *a* and the receiver *b*. The two derivations produce the following results.

```
proctype D_abp_D_a()
{
R0:     b!msg0 -> goto R1;
R1:     if
        :: a?ack0 -> goto R2
        :: a?ack1 -> goto R0
        fi;
R2:
        b!msg1 -> goto R1
}
proctype D_abp_D_b()
{
R0:     b?msg0 -> goto R1;
R1:     if
        :: a!ack0 -> goto R2
        :: a!ack1 -> goto R0
        fi;
R2:     b?msg1 -> goto R1
}
```

According to this specification the wrong acknowledgment may be repeated by the receiver and will be ignored by the sender. As a result, the skeleton state machine for *b* includes behavior that is permissible, but not desirable. To avoid this, we must rewrite the derived process, manually, as follows, splitting state R1 into two halves:

```
proctype D_abp_D_b()
{
R0:     b?msg0 -> goto R11;
R11:    if
        :: a!ack0 -> goto R2
        fi;
R12:    if
        :: a!ack1 -> goto R0
        fi;
R2:     b?msg1 -> goto R12
}
```

which can be simplified via

```
proctype D_abp_D_b()
{
R0:     b?msg0 -> goto R11;
R11:    a!ack0 -> goto R2;
R12:    a!ack1 -> goto R0;
R2:     b?msg1 -> goto R12
}
```

to its final form:

```
proctype D_abp_D_b()
{
R0:     b?msg0 -> a!ack0;
        b?msg1 -> a!ack1;
        goto R0
}
```

Now let us see how the derivation is affected if we expand the specification with a message to a third process *c* that logs all correctly transmitted and acknowledged messages with sequence number zero.

```
spec abp2
{
        do
        :: a->b!msg0;
                do
                :: b->a!ack0; a->c!log; break
                :: b->a!ack1; a->b!msg0
                od;
                a->b!msg1;
                do
                :: b->a!ack0; a->b!msg1
                :: b->a!ack1; break
                od
        od
}
```

The derivative of the specification for *c* is simply

```
proctype D_abp2_D_c()
{
R0:     c?log -> goto R0
}
```

The derivative for *b* remains unchanged, but the derivative for *a* becomes

```
proctype D_abp2_D_a()
{
R0:     b!msg0 -> goto R1;
R1:     if
        :: a?ack0 -> goto R2
        :: a?ack1 -> goto R0
        fi;
R2:     c!log -> goto R3;
R3:     b!msg1 -> goto R4;
```

```
R4:     if
        :: a?ack0 -> goto R3
        :: a?ack1 -> goto R0
        fi
}
```

10.3 DERIVATION ALGORITHM

The skeleton machine can be derived from a specification in two steps. First, if we derive a machine for process *p*, all messages in the specification that are not either sent or received by *p* are replaced by `skip`. Next, all specifications of the type

```
q->p!message
```

are translated into

```
p?message
```

and, similarly, all specifications

```
p->q!message
```

become

```
q!message
```

The last step is to handle cases such as these

```
R0:     if
        :: p?message0 -> goto R1
        :: skip -> goto R2
        fi
```

where the skip was inserted in the first step. In this case, an event outside the derived process can make the system change state, presumably changing the future behavior of the environment of the derived process. The derived process does not, and cannot, know when or if this invisible transition takes place. It must, however, be capable of accepting any incoming messages that may arrive after the invisible transition takes place. Therefore, for the above example, state `R0` of the derived process *inherits* all receive operations from state `R2`, together with the corresponding transitions.
If state `R2` is specified

```
R2:     p?message1 -> goto R3
```

the new state `R0` becomes

```
R0:     if
        :: p?message0 -> goto R1
        :: p?message1 -> goto R3
        fi
```

If state `R2` offers a choice

```
R2:     if
        :: p?message1 -> goto R3
        :: q!message2 -> goto R0
        :: p?message3 -> goto R0
        fi
```

we inherit only the receive operations and write

```
R0:     if
        :: p?message0 -> goto R1
        :: p?message1 -> goto R3
        :: p?message3 -> goto R0
        fi
```

The only remaining possibility is if state `R2` specifies only a send operation:

```
R2:     q!message2 -> goto R0
```

In this case the `skip` transition is omitted, and we write

```
R0:     if
        :: p?message0 -> goto R1
        fi
```

which simplifies to

```
R0:     p?message0 -> goto R1
```

This last case may be flagged as a potential inconsistency in the specification. The specification in this case requires a process to wait for an event that it cannot observe.

The last derivation step above is repeated until all the ‘‘hidden’’ transitions have been removed. Note that if the target state `R2` has its own `skip` transitions the last derivation step may require the inspection of still more states.

```
R2:     if
        :: p?message1 -> goto R3
        :: skip -> goto R0
        :: skip -> goto R4
        fi
```

The derivation algorithm can produce skeleton state machines for the target processes that adhere to the constraints of the specification. It illustrates one of the purposes of a protocol synthesis procedure: offering automated assistance to a protocol designer. The designer can concentrate on defining just one central item: the protocol specification itself.

Unfortunately, this design procedure gives no guarantee that the interaction of the derived processes will not lead to dynamic errors, such as deadlocks. Concentrating on that aspect of the design problem leads to a different type of design procedure, which we discuss next.

10.4 INCREMENTAL DESIGN

The following design method, originally published in 1980, is often used as a guideline for attempts to build protocol synthesis procedures. The procedure is interactive, and assumes the existence of an independent service specification that the designer will follow while developing the protocol processes.

The user specifies only message transmissions. The system deduces where in the protocol the corresponding receive actions are required. Initially, all processes, i.e., the ''skeleton state machines'' from the first method, are assigned a dummy initial state. The designer can now select one of the states in the system and extend it with a message transmission. The designer must specify the name of the message, its parameters, and its destination. For the process that is to transmit the message, the designer must also specify a successor state for the send action: either an existing or a newly created process state.

For each transmission edge added to one of the processes in this way, the synthesis software traces all possible states of the destination process in which the message can be received, and updates the state machine for that process automatically. The user has to name the successor states for all message receptions events that are added.

After each update, the incremental design procedure can warn the designer which *stable state tuples* have been created. A stable state tuple is defined as a composite system state in which no messages are in transit or stored in buffers. If such a composite system state is reachable, the state must persist until one of the processes sends a message. If none of the processes can transmit a message, the reachable stable state tuple corresponds to a deadlock.

The designer in this method can only specify send actions. The place where the corresponding receive actions are required is deduced by the synthesis software. This avoids unspecified receptions and certain types of deadlock, but it cannot guarantee the *functional* correctness of the protocol. That is, the synthesis method cannot guarantee that a protocol synthesized in this way will realize a given service.

10.5 PLACE SYNCHRONIZATION

The third approach can be considered a compromise between the first two methods discussed above. This method starts with a service specification written as a regular expression of synchronization requirements. The symbols in the service expression are the names of service primitives. The operators of the expression determine how the execution of these primitives is to be synchronized. In the expression

$$a^1;(b^2 \| c^3) \tag{1}$$

the superscripts denote service access points, the physical places where the service primitives are executed. The semicolon is used to indicate a sequential execution: the execution of service primitive *a*, at the place represented by 1, must have been completed before the primitives *b* or *c* can be executed at places 2 or 3, respectively. The parallel bars are used to indicate that the two subexpressions can be executed

simultaneously. Parentheses are used for grouping. A single bar between two subexpressions implies alternation, either one of the two subexpressions can be executed, but not both.

To enforce the synchronization requirements formalized in the service expression, the synthesis algorithm can derive a protocol that controls the execution of the service primitives. The sequential execution in expression (1), for instance, can be enforced by having the first primitive a^1 complete by transmitting a unique message from place 1 to places 2 and 3, and by delaying the execution of primitives b^2 and c^3 until that message has arrived.

The synthesis method is appealing, but it also has drawbacks. The method can derive protocols for only a limited class of global synchronization requirements. Not all protocol specifications can easily be expressed in those terms. Consider a reader/writer protocol for a data base shared between multiple processes. One method to secure the integrity of the data is to allow multiple readers to be active, but to allow access to at most one writer process at a time, and then only when no reader processes are active.

If a reader process *i*, requiring access to item *n*, is represented by the symbol r^{n_i}, and the corresponding writer process is represented by w^{n_i}, the design problem is now to write a regular expression on these symbols, using the operators ;, ||, and |. To properly describe the solution, we must count the number of active processes of each type and express the synchronization requirement as conditions on those counts. But the regular expression does not allow us to do that. If a synchronizing expression can be found, it may not be easier to find it than to invent the final protocol directly.

10.6 SUMMARY

An ideal method for protocol design would be to build a model from scratch that can be proven correct by construction. No such method exists, although many interesting attempts have been made. In this chapter we have given an overview of three such attempts. The first method allows one to extract skeleton state machines from a single, formalized statement of a correctness requirement. The method has drawbacks, the most important of which are:

- The method does not provide any help in the correct formalization of the protocol specification itself.
- The derived processes must, in some cases, be tuned to remove permissible but undesirable behavior. The method offers no help here, nor can it help us to verify that the alterations preserve the correctness of the derivations.

The second method interactively guides the protocol designer to a complete design and issues warnings on potential deadlocks. The most important drawbacks of this approach are:

- The method does not guarantee that the protocol constructed realizes a given service.
- The method does not guarantee absence of dynamic errors such as deadlocks. It can only warn for the possibility of a deadlock. When the number of possible

deadlock states rises, as it does in a design of a realistic size, it quickly becomes impossible for a human designer to verify manually that all potential deadlock states are effectively unreachable.

The third method derives protocols from concise expressions of global synchronization requirements. Its main drawback is:

- Only a restricted class of protocol design problems can be expressed in the regular expression language on which the method is based.

All three methods share one other drawback that is perhaps of even greater importance: they do not really seem to facilitate the design process.

EXERCISES

10-1. Try to derive Lynch's protocol (Chapter 2) and parts of the file server protocol (Chapter 7) with a protocol synthesis method. □

10-2. Some protocol synthesis methods that have been described in the literature guarantee ''correctness by construction'' with the help of an exhaustive reachability analysis algorithm that is run over partial specifications during the design. Consider the possible drawbacks of this method. □

10-3. Compare the place synchronization method with the protocol derivation method. Both start out with an abstract ''service specification'' from which a protocol is derived. How do the two types of service specifications differ? Do they have the same expressive power? □

10-4. Develop a workable protocol synthesis method and mail the solution to the author for the next edition of this book. □

BIBLIOGRAPHIC NOTES

This chapter has given only a brief overview of synthesis methodologies since, alas, none exist that can adequately solve the protocol design problem.

The best known method for protocol synthesis is the incremental method from Section 10.4. It was first described in Brand and Zafiropulo [1980]. The method has many variations and has even been applied in protocol validation algorithms. The place synchronization method from Section 10.5 is a formal method to derive parts of a lower-level protocol from a higher-level service specification. The method is fully developed in Gotzheim and Bochmann [1986]. A variant can also be found in Chu and Liu [1988].

The derivation method from Section 10.2 can be seen as an extension of earlier work on methods to derive the description of a protocol entity from a specification of its communication partner, see Zafiropulo et al. [1980], Gouda [1983], Merlin and Bochmann [1983].

Not studied here is a potentially interesting, recent application of control theory to the protocol synthesis problem that was reported in Rudie and Wonham [1990]. In this approach, the original protocol system is first described as an uncontrolled process in which all feasible actions, such as message transfers, happen chaotically. A high-level service specification details the constraints for the system. Assuming that the

process contains a sufficient number of control points, a protocol can then be derived as a minimal restriction of the chaotic process behavior that satisfies the system constraints.

Several methods have also been studied for partitioning a sequential program into a distributed program, preserving functionality and correctness, e.g., Moitra [1985], Prinoth [1982]. These algorithms require an initial solution to the problem, through the derivation of a sequential program, before the synthesis method itself can be applied.

A general overview of protocol synthesis methods can be found in Chu [1989].