

Chapter 2

THE TIMED ABSTRACT PROTOCOL NOTATION

The Timed Abstract Protocol notation, or TAP, is designed to be a small language for describing asynchronous, message passing network protocols. This focus entails several features:

- TAP is intended to describe protocols which normally wait for an externally generated event such as a received message or a timeout, then perform local computation to handle the event. As a result, the processes in TAP are made up of guarded actions, where the guards are based on the availability of a received message, or a timeout occurrence as well as local state.
- Network protocols generally deal with two kinds of information: *protocol control information*, which is mainly made up of integral values and process addresses, and *data*, which is uninterpreted by the protocol. Also, the processing of a protocol at a local level is usually simple and should terminate quickly. Following these two observations, the state of processes is limited to minimal control information and the control structures are kept simple. These limitations serve to ease understanding and verification.
- Specialized operations, such as sending a message, are necessary for asynchronous message passing protocols. These operations are integrated into the language where they have specific semantics, again easing understanding and verification.

In TAP, a network protocol consists of two or more processes, which communicate by sending messages across channels. In this chapter, we begin to present the syntax and general semantics of TAP by discussing a simple example protocol.

Messages and channels

In TAP, a message is a sequence of fields, where each field is either an integral value consisting of protocol control information or a byte array containing uninterpreted data. The integral fields can be either constant or variable; constant fields can be used to distinguish messages while variable fields carry protocol information. For example, Figure 2.1 shows two messages, each of which has a single, constant field identifying it. Figure 2.1 does not show the full range of options; variable fields would not specify values and data fields would specify the field size in bytes, with either a constant or non-constant expression.

message rqst	message rply
begin	begin
type : 8 bits = 0	type : 8 bits = 1
end	end

Figure 2.1: Request/reply messages.

A channel is a queue of messages. Messages are inserted into the tail of the queue by the sending process and removed from the head by the receiving process.

When discussing a protocol, the contents of a channel are referred to by a sequence of messages surrounded by angle brackets and separated by semicolons: $\langle m;n \rangle$. A channel itself is referred to by the notation **ch.p.q**, where **p** and **q** are the names of the sending process and the receiving process, respectively.

In the TAP notation, channels are implicit—they are identified by the abstract addresses of the sending and receiving processes. Since each process is itself one of the endpoints, only the remote address is needed within a process; the name of the process identifies the address of its local endpoint.

Every process is connected to every other process by channels in both directions. As a result, every process can send messages to any other process it has the address of.

Processes

A process consists of a local state and a set of actions describing the behavior of the process. Figure 2.2 shows the two processes of a simple request/reply protocol.

<pre> process p const q : address var readyp : boolean = true begin readyp → send rqst to q; readyp := false rcv rply from q → readyp := true end </pre>	<pre> process q var p : address begin rcv rqst from p → send rply to p end </pre>
---	--

Figure 2.2: Request/reply protocol, version 1.

The local state of a process is described by variables and constants, of one of a number of data types. There are two data types used in processes p and q in Figure 2.2:

1. An **address** is used to identify a channel between the current process and another, either to send a message to another process or to recognize the sender of a received message.¹
2. A **boolean** is either true or false.

The remaining basic data type, **integer**, has values from 0 to $2^{32} - 1$. The syntax of TAP also allows smaller ranges of integers as well as multidimensional arrays of any of these data types.

¹An address is special in that it cannot be assigned a literal value in TAP. Instead, its value must be provided by the environment of the process. As described later, variable addresses receive values when used in receive guards. Additionally, the Austin Protocol Compiler runtime system provides the capability to set addresses outside the executable code generated by the compiler.

Actions

Actions describe the computation performed by a process. Each action consists of a guard, describing the circumstances under which the action is enabled, and a body, providing the statements which are executed when the action is executed.

When discussing protocol computations, the individual actions are referred to by the notation $p.n$, where p is the name of the process and n is the number of the action within the process. Similarly, $p.id$ refers to a variable or constant id in a process p .

There are two kinds of action guards in Figure 2.2:

1. A local guard is a Boolean predicate involving the local state of the process. The action is enabled when the predicate is true. An example of a local guard is that of $p.1$, the first action of process p in Figure 2.2. The guard of this action is the predicate `ready p` .
2. A receive guard identifies a message and the address of a remote process and is enabled when a matching message is at the head of the channel from a remote process. An example is $p.2$, the second action of process p . The guard of this action is `rcv rply from q` , which is enabled when a `rply` message is at the head of the channel from q to p . When this action is selected for execution, as discussed later, the `rply` message will be removed from the channel.

Again, the address here is special—a receive guard with a constant address will only be enabled when the matching message is at the head of the channel from the process identified by the address. On the other hand, a receive guard with a variable address is enabled when the matching message is at the head of the channel from *any* process; during the execution of the action, the address variable takes as its value the address of the remote process.

Statements

Most of the statements in TAP are relatively conventional. There are two kinds of statements in Figure 2.2:

- The first is a send statement, **send** *rqst* **to** *q*. This statement inserts the message *rqst* into *ch.p.q*.
- The second is an assignment statement, such as *readyp* := **false**. TAP supports multiple assignment, where a list of variables on the left-hand-side is matched by a list of expressions on the right-hand-side.

When a message is received or sent by an action, it introduces a message structure that is local to the action. This message structure has records labeled by the field names from the message. When a message is received, the records contain the values of the fields from the incoming message. The values of the fields of messages to be sent can be set by assigning to the records before sending the message. These structures, however, do not participate in the state of the process because they are not preserved between actions.

The statements are sequentially combined by separating them with a semicolon.

Additional statements are also available in TAP:

- The **skip** statement does nothing.
- A conditional statement,

$$\mathbf{if} \ p_0 \rightarrow s_0 \ \|\ p_1 \rightarrow s_1 \ \|\ \dots \ \|\ p_n \rightarrow s_n \ \mathbf{fi},$$

chooses a branch $p_i \rightarrow s_i$ nondeterministically from the branches with true predicates among p_0, \dots, p_n and executes the corresponding statement s_i .

- The iteration statement, **do** $p \rightarrow s$ **od**, executes statement s repeatedly, as long as predicate p is true.

Protocol style

A number of issues apply to the design of protocols specified in the TAP notation. One is illustrated in Figure 2.2: protocol quiescence. Another two are loop termination, and conditional completeness.

The protocol should be quiescent: it should not be possible for any process in the protocol to continue executing local actions indefinitely. An action, in

TAP, is executed *when* its guard is enabled; this is at variance with AP, where an action is executed *only when* its guard is enabled. The difference is that a process in TAP cannot wait when an action is enabled. For example, if an action sending a message has a **true** local guard, the action will flood the network with instances of mesasge it sends.

Instead, every process in the protocol should only execute a finite number of local actions before every local action becomes disabled—this prevents the process from abusing local resources. Such a process can instead wait for a message to be received or for a timeout to expire.

A related issue is loop termination. Although neither process *p* nor process *q* needs a **do** statement, other protocols will, and the necessity that actions be atomic requires that loops terminate deterministically, and preferably after performing only the computations needed by the protocol. This prevents the execution of a single action from blocking all of the other computations of a process.

Finally, the conditional statement syntactically uses multiple branches, each with a boolean predicate and a body of statements. For clarity, the predicates should be mutually exclusive, so that a single branch is possible in any state, and the disjunction of the predicates should be true, so that some branch is taken from any state.

Justification

The TAP notation is intended to specify message-passing network protocols simply and clearly. Each of the features of the notation supports this goal:

1. The message specification describes the minimal features needed for most common messages of existing protocols. It is neither as complete nor as complex as other message specification languages such as ASN.1[45] and XDR[46]. Instead, the TAP message notation is restricted to describing a simple format which is interoperable with many Internet protocols while allowing the programmer the flexibility to deal with other messages.
2. Processes have a simple structure, particularly in the definition of their local state and the limited set of actions used to describe their behavior.

Coupled with the execution model, described in the next chapter, these features make reasoning about network protocols easier.

3. Like Promela[39], Teapot[27], Esterel[40], and other formal notations, TAP tries to avoid fine-grained data manipulation while expressing overall control structure. The statements describing the behavior of actions are limited in number and simplified, when compared with general purpose languages. This simplification, first, eases reasoning about the behavior, and second, reduces the tendency to include overly complex behavior or behavior unrelated to the network protocol in the specification.

This chapter has only described the basic syntax and semantics of TAP. The details of TAP computations are the subject of the next chapter. However, the next section contains a detailed discussion of the TAP language, based on its grammar. Understanding the next section is not necessary in order to understand subsequent chapters.

Details of TAP

In previous sections, we examined the TAP notation from an abstract viewpoint, as a formal notation for specifying network protocols. Much of the remainder of this work will continue with that viewpoint, but this section examines the TAP notation as a programming language.

The discussion of the complete of the TAP language follows the structure of the TAP grammar—it re-covers the parts of TAP described previously and includes features of TAP that will not be described fully until the next chapter as well as features that are not further mentioned. The grammar is described using the Extended Backus-Naur Format, with the following conventions:

- {...} indicates zero or more copies of the contained elements.
- [...] indicates zero or one copy of the contained elements; i.e. the contents are optional.
- (...|...) indicates a choice between the contained elements.
- Literal text is presented in quotation marks.

- Non-literal token elements are in italics. There are three of these:
 - A *string* is a quote-delimited string of characters which does not span lines. Internal quotes and newlines can be escaped by a backslash. These strings cannot be manipulated in TAP, but can serve as arguments to functions as well as to directives as described later.
 - A *number* is one or more decimal digits, indicating a non-negative number.
 - An *id* is an identifier, made up of a letter followed by any number of letters or numbers.

Parsing of each source file begins with the start symbol:

```

start ::= elements
elements ::= {element}
element ::= "import" string
            | "include" string
            | message
            | process

```

The source file given to the compiler consists of a sequence of elements. Each element is either an import directive, an include directive, a message definition, or a process definition.

The import directive looks for the file named in the *string*. The contents of this file are read and processed by the compiler before any subsequent elements in the current source file.

The include directive inserts a C include directive in the output file, calling for the file named by the *string*. These included files form part of the interface between the APC-generated C module and external C code.

Message syntax

```

message ::= m-header m-body
m-header ::= ["external"] "message" m-name [m-functs]
m-name ::= id
m-functs ::= "(" m-in "," m-out ")"
m-in ::= id

```


m-out ::= *id*
m-body ::= “**begin**” fields “**end**”
fields ::= { field “,” } field

Each message definition consists of a header and a body. The message header primarily provides a name for the message. The body of the message is a sequence of fields, separated by commas. The message definition is used by the compiler to produce:

1. A C structure with records for each field in the message.
2. Parsing and marshalling functions, which interpret and recognize received messages and convert a message structure to a sequence of bytes for transmission, respectively.

Optionally, the message can be marked as external, in which case the compiler does not generate the C functions for marshalling and parsing the message. This allows the programmer to provide such functions, in order to handle more complex messages than those that can be described by TAP. Also optionally, two functions can be identified which process the message immediately after the fields in the message have been parsed (m-in) and immediately before the message is sent (m-out). These functions receive the message buffer as well as the structure describing the fields of the message, allowing them to compute a checksum for the message, for example.

field ::= *f-name* “:” *f-type* [“=” *f-value*]
f-name ::= *id*
f-value ::= *expression*

Each field definition consists of a field-name, a field-type, and optionally, a field-value. If the field-value is present, the field is considered constant; the field is automatically set to that value before the message is sent and received messages are checked to ensure the field contains the proper value as part of the process of recognizing messages. In these expressions, the only allowable values are constants and the names of previous fields.

f-type ::= *f-size* (“**bits**” | “**bytes**”)
f-size ::= *expression*

A field-type describes the size and type of the contents of the field. The expression describing the size can contain literal values or the names of previous fields in the message. The type of the field is implied by the use of bits or bytes to describe the field.²

- A bit field contains an unsigned integer value. The size expression describes the size of the field in bits; it must not be larger than 32 bits.
- A byte field contains a sequence of data bytes. The size expression describes the size of the field in 8-bit bytes. For a received message, the value of the record for the field in the structure generated by the compiler will be a pointer to the data in the original message buffer. When building a message to be sent, the value of the record should be set to a pointer to a sequence of bytes which will remain valid until the message is sent.

Each message has an additional field, named *size*, which indicates the overall size of the message in bytes. When receiving a message terminating with an arbitrary-length data field, the *size* field (minus the size of any previous fields) provides the length of the final field. When sending such a message, assigning to the *size* field allows the message marshalling functions to copy the appropriate number of bytes from the array pointed to by the data field.

Process syntax

```

process ::= p-header p-body
p-header ::= "process" p-name [constants] [variables]
p-name ::= id
constants ::= "const" declarations
variables ::= "var" declarations
declarations ::= {declaration ";" } declaration
p-body ::= "begin" actions "end"

```

Each process definition also consists of a header and a body. The process header provides a name for the process as well as the optional declarations for

²For grammatical correctness, “**bit**” is allowed as a synonym for “**bits**” and likewise, “**byte**” for “**bytes**”.

the process's constants and variables. The process's body contains a sequence of actions.

```

declaration ::= ids ":" type ["=" initial-value]
ids ::= { id "," } id
type ::= "integer"
           | number ".." number
           | "boolean"
           | "address"
           | "array" "[" array-size "]" "of" type
array-size ::= number
initial-value ::= (number | "true" | "false")

```

In each declaration, a sequence of identifiers which name constants or variables are associated with a type and optionally an initial value. The basic types allowed by TAP are 32-bit integers, booleans, and addresses. The integer type can be specified as either a general integer or as a range of allowed values.

The initial values for variables or constants must match the type of the variable or constant; the value of an integer is a number, and the value of a boolean is either true or false. Addresses may not be given an initial value in TAP. (The initial value of an address can be given via the C interface while initializing the APC runtime system. See Chapter 6 for more information.)

The only complex type supported by TAP is the array, with any number of dimensions. The allowed indices of each array dimension is given by the array-size value; indices range from 0 to the array-size-1. If an initial value is given for an array, each element of the array is set to the value.

Action syntax

```

actions ::= { action "[" ]" } action
action ::= guard "->" statements
guard ::= (local-guard | receive-guard | timeout-guard)
local-guard ::= expression
receive-guard ::= "rcv" m-name "from" address
address ::= id
timeout-guard ::= "timeout" t-name

```

t-name ::= *id*

In a TAP process, actions are separated by a box, written as two square brackets: []. Each action consists of a guard and a sequence of statements. There are three forms of guards: local, receive, and timeout. Chapter 3 describes the behavior of each of the guards in more detail, and Chapter 6 contains the details of the runtime support for each guard.

Local guards are made up of a predicate, a boolean expression. The action is enabled when the guard evaluates to true.

Receive guards specify a message accepted by the action and an address. The guard may be enabled if and only if the received message matches the message specified by the receive guard. If the address is a constant, then the action will only be enabled if the message is from the process identified by the address. If the address is a variable, then the action will be enabled no matter where the message is from and the address will be set to the source of the message.

Timeout actions provide a name, t-name, for the action for use with the activation statement; the behavior of such actions is described in the next chapter.

Statement syntax

statements ::= {statement “;”} statement
statement ::= “skip” | function-call | assignment | send
| conditional | loop | activate

In any sequence of statements, the individual statements are separated by semicolons. The two fundamental statements are skip, which does nothing, and a function call, which invokes a C function and is more fully described on page 28.

assignment ::= left-sides “:=” expressions
left-sides ::= {left-side “,”} left-side
left-side ::= (*id* | field-reference | array-reference)
expressions ::= {expression “,”} expression

TAP assignment statements allow multiple values to be assigned simultaneously; in the code generated by the APC compiler, each expression is evaluated

independently and stored in a temporary location. Subsequently, the values are assigned to the left-hand-side locations. Locations which can be assigned values are either variables, message fields, or array elements.

send ::= “send” m-name “to” address

A fundamental operation in TAP is sending a message, identified by m-name, to a process, identified by the address. Any necessary fields in the message should be set before executing the send statement.

conditional ::= “if” guarded-statements “fi”

guarded-statements ::= { guarded-statement “[]” } guarded-statement

guarded-statement ::= expression “->” statements

TAP provides a conditional statement with guarded branches separated by the box. Each branch consists of a boolean expression guarding a sequence of statements. In execution, one branch with a true-valued expression is chosen and executed. If no branches are enabled, execution continues with the next statement after the conditional.

loop ::= “do” expression “->” statements “od”

The iteration statement in TAP is made up of a single guarded statement, which provides a sequence of statements which are executed repeatedly as long as the expression evaluates to true.

activate ::= “act” t-name “in” delay

delay ::= expression

The activate statement, along with the timeout guards, is discussed in detail in Chapter 3. In general terms, it sets a timer associated with the timeout guard identified by t-name. The delay gives the value of the timer³, after which the timeout guard enables the corresponding action.

³In milliseconds.

Expression syntax

In order to simplify the description of the TAP expression, the grammar rule is broken into a number of sub-rules below. The expression rule is the combination of all of the individual sub-rules.

expression ::= (*id* | *number* | “true” | “false” | *string*)

The fundamental expressions in TAP are variable names, numbers, true and false, and strings (which may only be used as arguments to function calls).

**expression ::= field-reference
 | array-reference
 | function-call
field-reference ::= m-name “.” (f-name | “size”)
array-reference ::= (array-reference | *id*) “[” expression “]”
function-call ::= function-name “(” [expressions] “)”
function-name ::= *id***

Further expressions are field references, array references, and function calls. Field references are described by a message name and either a field within the message or the special field, “size”, which contains the overall size of the message in bytes.

Array references follow the traditional syntax, with a numeric expressions describing the element within the array.

A function call identifies a C function by name and executes it with the arguments given by the expressions. The C type of the return value of the function should be one of:

- void, for functions called as statements,
- unsigned long, for integer values, or
- unsigned char *, for an assignment to a message’s data field.

**expression ::= “(” expression “)”
 | expression binary-operator expression
 | unary-operator expression**

binary-operator ::= “=” | “>” | “<” “<=” | “>=” | “<>”
 | “|” | “&” | “+” | “-” | “*” | “/”
unary-operator ::= “~” | “-”

The next group of general expressions include the normal binary and unary operators. The binary operators are equality, inequality, boolean operators, and arithmetic operators. Unary operators are boolean and arithmetic negation.

5	“=” “>” “>=” “<” “<=” “<>”
4	“&” “ ”
3	“+” “-”
2	“*” “/”
1	“~” “-” (unary)

Figure 2.3: TAP operator precedence, from lowest to highest.

These operators have the precedence described in Figure 2.3.

expression ::= “**size**”

The final form of expression, a bare reference to a size message field, is only valid in an expression that is part of a message definition. The value of the “**size**” expression is the overall size of the message in bytes.



<http://www.springer.com/978-0-387-23227-0>

The Austin Protocol Compiler

McGuire, T.M.; Gouda, M.G.

2005, XIII, 141 p., Hardcover

ISBN: 978-0-387-23227-0