Embedded Systems Specification and Design
Model-based Design and Verification

David Kendall

Tuples

In TLA$^+$, a tuple is any function whose domain is a set 1..*N* for some natural number *N*

Function application is denoted in the usual way, e.g. $t[i]$ refers to the *i*th component of tuple *t*

A tuple can be written using $\langle \ldots \rangle$ (`<< ... >>` in ASCII)

Let $t = \langle$ *"Season", "of", "mists", "and", "mellow", "fruitfulness"* $\rangle$ be a tuple of strings, then $t[3] =$ *"mists"* and $t[6] =$ *"fruitfulness"*. $t[7]$, $t[0]$, and $t[-57]$ are unspecified.

Sets of tuples can be written with the *Cartesian product* operator, $\times$ (`\X` in ASCII)

For sets *A* and *B*, $A \times B$ is the set containing all tuples *t*, where DOMAIN $t = 1 \,..\, 2$, and $t[1] \in A$ and $t[2] \in B$, e.g.

$$\{\text{"a"}, \text{"b"}, \text{"c"}\} \times \{5, 6\} = \{\langle \text{"a"}, 5\rangle, \langle \text{"a"}, 6\rangle, \langle \text{"b"}, 5\rangle, \langle \text{"b"}, 6\rangle, \langle \text{"c"}, 5\rangle, \langle \text{"c"}, 6\rangle\}$$

Sequences

TLA$^+$ treats tuples as finite sequences

A sequence is just like a list in a programming language such as Python

The standard module, *Sequences*, defines some useful operators on sequences

$\mathit{Seq}(S)$	The set of all sequences of the set (S), e.g. $\langle 3, 7, 1, 9 \rangle$ is an element of $\mathit{Seq}(\mathit{Nat})$ (sequences of natural numbers)
$\mathit{Head}(s)$	The first element of sequence s, e.g. $\mathit{Head}(\langle 3, 7, 1, 9 \rangle)$ is 3
$\mathit{Tail}(s)$	The tail of sequence s, which consists of s with its first element removed, e.g. $\mathit{Tail}(\langle 3, 7, 1, 9 \rangle)$ is $\langle 7, 1, 9 \rangle$
$\mathit{Append}(s, e)$	The sequence obtained by adding element e to the end of sequence s, e.g. $\mathit{Append}(\langle 7, 1, 9 \rangle, 3)$ is $\langle 7, 1, 9, 3 \rangle$
$\mathit{Len}(s)$	The length of sequence s, e.g. $\mathit{Len}(\langle 3, 7, 1, 9, 3, 3 \rangle)$ is 6

Records

TLA$^+$ provides some convenient syntax for particular kinds of function, which it calls *records*

Records in TLA$^+$ are just functions whose domain is a finite set of strings

Programmers can think of TLA$^+$ records as being like dictionaries in Python or structs in C

For example, details about a person might be represented by a record:

$$p = [name \mapsto \textit{"Alan Shearer"}, age \mapsto 48]$$

This is a function with domain $\{\textit{"name"}, \textit{"age"}\}$

Instead of writing $p[\textit{"name"}]$ or $p[\textit{"age"}]$, we can write $p.name$ and $p.age$ to refer to the *fields* (components) of the record, p

A notation for describing the set of all possible person records is

$$[name : String, age : 0..150]$$

Sequences and records : An example

A simple scheduler

MODULE *SimpleScheduler*

EXTENDS *Naturals*, *Sequences*

CONSTANT
  PROCID,
  PRIORITY

VARIABLE
  active,
  procs,
  running,
  ready,
  blocked

$Process \triangleq [priority : PRIORITY, state : \{\text{“running”}, \text{“ready”}, \text{“blocked”}\}]$

$IDLE \triangleq \text{CHOOSE } p : p \notin PROCID$

$TypeOk \triangleq$

$$
\begin{aligned}
&\land active \subseteq PROCID \\
&\land procs \in [active \rightarrow Process] \\
&\land running \in active \cup \{IDLE\} \\
&\land ready \in Seq(active) \\
&\land blocked \in Seq(active)
\end{aligned}
$$

A simple scheduler (ctd)

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ active = \{\} \\
\quad \land\ procs = \langle\rangle \\
\quad \land\ running = IDLE \\
\quad \land\ ready = \langle\rangle \\
\quad \land\ blocked = \langle\rangle
\end{array}
$$

$$
\begin{array}{l}
Create(p) \triangleq \\
\quad \land\ p \in PROCID \setminus active \\
\quad \land\ active' = active \cup \{p\} \\
\quad \land\ procs' = [pp \in active \cup \{p\} \mapsto \\
\qquad\qquad\qquad \text{IF } pp \in \text{DOMAIN } procs \\
\qquad\qquad\qquad\quad \text{THEN } procs[pp] \\
\qquad\qquad\qquad\quad \text{ELSE } [priority \mapsto p,\ state \mapsto \text{“ready”}]] \\
\quad \land\ ready' = Append(ready,\ p) \\
\quad \land\ \text{UNCHANGED } \langle running,\ blocked\rangle
\end{array}
$$

$$
\begin{array}{l}
Run(p) \triangleq \\
\quad \land\ running = IDLE \\
\quad \land\ ready \neq \langle\rangle \\
\quad \land\ p = Head(ready) \\
\quad \land\ running' = p \\
\quad \land\ procs' = [procs \text{ EXCEPT } ![p] = [priority \mapsto procs[p].priority,\ state \mapsto \text{“running”}]] \\
\quad \land\ ready' = Tail(ready) \\
\quad \land\ \text{UNCHANGED } \langle active,\ blocked\rangle
\end{array}
$$

A simple scheduler (ctd)

$Yield(p) \triangleq$
$\quad \land \ running = p$
$\quad \land \ running' = IDLE$
$\quad \land \ procs' = [procs \text{ EXCEPT } ![p] = [priority \mapsto procs[p].priority, \ state \mapsto \text{"ready"}]]$
$\quad \land \ ready' = Append(ready, \ p)$
$\quad \land \ \text{UNCHANGED } \langle active, \ blocked \rangle$

$Terminate(p) \triangleq$
$\quad \land running = p$
$\quad \land running' = IDLE$
$\quad \land active' = active \setminus \{p\}$
$\quad \land procs' = [pp \in \text{DOMAIN } procs \setminus \{p\} \mapsto procs[pp]]$
$\quad \land \text{UNCHANGED } \langle ready, \ blocked \rangle$

$Next \triangleq$
$\quad \exists \, p \in PROCID :$
$\qquad \lor Create(p)$
$\qquad \lor Run(p)$
$\qquad \lor Yield(p)$
$\qquad \lor Terminate(p)$

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

Pluscal

Pluscal is a language for writing formal specifications of algorithms

It is like a very simple programming language except:

- Any TLA$^+$ expression can be used in a Pluscal algorithm
- Pluscal can represent non-determinism
- A Pluscal algorithm appears as a comment in a TLA$^+$ module and is then translated automatically into TLA$^+$
- The model checker TLC can be used to check properties of Pluscal algorithms

Pluscal is convenient for expressing the flow of control in sequential and shared-memory concurrent algorithms

Pluscal offers two slightly different syntaxes: the p-syntax and the c-syntax. You should use the p-syntax for all of your work in this module

The Die Hard Problem

Given a tap and 2 unmarked jugs, one with a capacity of 5 gallons and the other with a capacity of 3 gallons, reach a state in which the larger of the two jugs contains exactly 4 gallons

You are allowed to iterate the following actions:

- fill either of the two jugs using the tap
- empty either of the two jugs by pouring its contents away
- pour the contents of one jug into the other jug, either filling the receiving jug or emptying the dispensing jug

Let's solve the Die Hard problem by writing a Pluscal algorithm

Die Hard in Pluscal

```
--algorithm PDieHard
  variable
    big = 0;
    small = 0;
  begin
    while TRUE do
      either big := 5                                    \* Fill big
          or small := 3                                  \* Fill small
          or big := 0                                    \* Empty big
          or small := 0                                  \* Empty small
          or with poured = Min(5 - big, small) do        \* small to big
               big := big + poured;
               small := small - poured
             end with
          or with poured = Min(3 - small, big) do        \* big to small
               small := small + poured;
               big := big - poured
             end with
      end either
    end while
  end algorithm
```

Pluscal statements

assignment The assignment statement is written with `:=` and has its usual meaning, e.g. `big := 5` assigns the value 5 to the variable `big`. The symbol = is used to test equality and to initialise variables. It is not used for assignment

while The `while` statement introduces a loop, in the usual way, e.g.

```
while someTest do
  body
end while
```

repeatedly tests the condition `someTest` and, if it is TRUE, executes the `body` of the statement. The loop terminates when `someTest` is FALSE

Pluscal statements (ctd)

either The `either` statement has the form:

```
either clause_1
    or clause_2
    ...
    or clause_n
end either
```

It is executed by non-deterministically executing any `clause_i` that is executable, and executing it

with The statement `with id \in S do body end with;` is executed by executing the statement sequence `body` with `id` equal to a non-deterministically chosen value of `S`. Execution is not possible if `S` is empty. The statement `with id = expr do ...` is equivalent to `with id \in {expr} do ...`

TLA$^+$ translation of Pluscal Die Hard

```
\* BEGIN TRANSLATION
VARIABLES big, small

vars == << big, small >>

Init == (* Global variables *)
        /\ big = 0
        /\ small = 0

Next == \/ /\ big' = 5
           /\ small' = small
        \/ /\ small' = 3
           /\ big' = big
        \/ /\ big' = 0
           /\ small' = small
        \/ /\ small' = 0
           /\ big' = big
        \/ /\ LET poured == Min(5 - big, small) IN
                /\ big' = big + poured
                /\ small' = small - poured
        \/ /\ LET poured == Min(3 - small, big) IN
                /\ small' = small + poured
                /\ big' = big - poured
\* END TRANSLATION
```

Concurrency in Pluscal

Reasoning about the behaviour of concurrent processes is one of the most significant challenges in designing distributed and/or embedded systems

Pluscal helps to simplify the task of modelling concurrent systems in TLA$^+$

A Pluscal description of a multi-process system can be translated automatically into TLA$^+$

The TLC model checker can be used to analyse the behaviour of the TLA$^+$ translation

Bank Account: A simple 2-process algorithm

Let's model a simple system in which 2 processes withdraw money from a bank account

The bank account is modelled just by its `balance`, which is initially £100

Each process withdraws £10 and then stops. The final balance should be £80

```
--algorithm BankAccount
    variable balance = 100;

    process Withdraw10 \in (1..2)
      variable current = 0;
    begin
s1:   current := balance;
s2:   current := current - 10;
s3:   balance := current;
    end process
end algorithm
```

Notes on the Pluscal Bank Account

A multi-process algorithm is introduced and concluded in the usual way

```
--algorithm SomeName

end algorithm
```

Processes are introduced with the keyword `process`. Each process has a name (not necessarily unique) and an identifier (unique), taken from the same set as all other processes. A process can introduce local variables, e.g.

```
process Withdraw10 \in (1..2)
variable

begin

end process
```

This introduces 2 processes called `Withdraw10` and identified by elements in the set $\{1, 2\}$

TLA$^+$ translation of Pluscal Bank Account

```
\* BEGIN TRANSLATION
VARIABLES balance, pc, current

vars == << balance, pc, current >>

ProcSet == (1..2)

Init == (* Global variables *)
        /\ balance = 100
        (* Process Withdraw10 *)
        /\ current = [self \in 1..2 |-> 0]
        /\ pc = [self \in ProcSet |-> "s1"]

s1(self) == /\ pc[self] = "s1"
            /\ current' = [current EXCEPT ![self] = balance]
            /\ pc' = [pc EXCEPT ![self] = "s2"]
            /\ UNCHANGED balance

s2(self) == /\ pc[self] = "s2"
            /\ current' = [current EXCEPT ![self] = current[self] - 10]
            /\ pc' = [pc EXCEPT ![self] = "s3"]
            /\ UNCHANGED balance

s3(self) == /\ pc[self] = "s3"
            /\ balance' = current[self]
            /\ pc' = [pc EXCEPT ![self] = "Done"]
            /\ UNCHANGED current

Withdraw10(self) == s1(self) \/ s2(self) \/ s3(self)

Next == (\E self \in 1..2: Withdraw10(self))
           \/ (* Disjunct to prevent deadlock on termination *)
              ((\A self \in ProcSet: pc[self] = "Done") /\ UNCHANGED vars)
\* END TRANSLATION
```

Notes on the TLA$^+$ translation

The global and local variables are all introduced in the same `VARIABLES` section

A definition, `ProcSet`, is introduced to represent the process identifiers

The local variables are modelled by functions from process identifiers to variable values, i.e. each process has its own copy of the local variables, indexed by process identifier

The *labels*, `s1`, `s2`, and `s3`, are used to identify the actions (steps) of the process. Each step is translated into its own TLA$^+$ operation. The placement of labels defines the *granularity* of the steps and is crucial to the behaviour of processes

The TLA$^+$ translation introduces a local variable, `pc` (program control), for each process, to indicate the current step in its execution

For a process that terminates, a label, `Done`, is introduced to model termination

Each process is modelled as the disjunction of its steps

The `Next` relation is modelled as the disjunction of the processes

Using TLC to check the BankAccount model

For a bank account with an initial balance of £100, we expect the final balance to be £80, after the completion of 2 withdrawals of £10 each

We can use TLC to check that this is the case for the bank account that we have modelled in Pluscal

Termination of the processes is indicated when the value of `pc` for each process is `Done`

We can define a `BalanceOk` property as follows:

```
BalanceOk == (pc[1] = "Done" /\ pc[2] = "Done") => balance = 80
```

We can use TLC to check if this property is invariantly TRUE. If it is not TLC will show a behaviour, starting in the initial state, that reaches a state in which the alleged invariant is FALSE

Demo Show the use of TLC to check this invariant.