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 \(\langle \ldots \rangle\) (\(<< \ldots >>\) 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 \textit{Cartesian product} operator, \(\times\) (\(\setminus x\) in ASCII)

For sets \(A\) and \(B\), \(A \times B\) is the set containing all tuples \(t\), where \(\text{DOMAIN } t = 1 .. 2\), and \(t[1] \in A\) and \(t[2] \in B\), e.g.

\[
\{ "a", "b", "c" \} \times \{ 5, 6 \} = \\
\{ \langle "a", 5 \rangle, \langle "a", 6 \rangle, \langle "b", 5 \rangle, \langle "b", 6 \rangle, \langle "c", 5 \rangle, \langle "c", 6 \rangle \}
\]
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

Seq(S) The set of all sequences of the set (S), e.g. \langle 3, 7, 1, 9 \rangle is an element of Seq(Nat) (sequences of natural numbers)

Head(s) The first element of sequence s, e.g. Head(\langle 3, 7, 1, 9 \rangle) is 3

Tail(s) The tail of sequence s, which consists of s with its first element removed, e.g. Tail(\langle 3, 7, 1, 9 \rangle) is \langle 7, 1, 9 \rangle

Append(s, e) The sequence obtained by adding element e to the end of sequence s, e.g. Append(\langle 7, 1, 9 \rangle, 3) is \langle 7, 1, 9, 3 \rangle

Len(s) The length of sequence s, e.g. 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:

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

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

Instead of writing \(p["name"]\) or \(p["age"]\), 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

\[
\{ \text{name} : \text{String}, \text{age} : 0..150 \}\]
Sequences and records: An example
A simple scheduler

```latex
\[
\begin{aligned}
\text{module SimpleScheduler} & \text{ extends Naturals, Sequences} \\
\text{constant} & \text{ PROCID, PRIORITY} \\
\text{variable} & \text{ active, procs, running, ready, blocked} \\
\text{Process} & \triangleq \{ \text{priority : PRIORITY, state : \{"running", "ready", "blocked"\}} \} \\
\text{IDLE} & \triangleq \text{choose } p : p \notin \text{Process} \\
\text{TypeOk} & \triangleq \\
& \land \text{active } \subseteq \text{PROCID} \\
& \land \text{procs } \in [\text{active } \rightarrow \text{Process}] \\
& \land \text{running } \in \text{active } \cup \{\text{IDLE}\} \\
& \land \text{ready } \in \text{Seq(active)} \\
& \land \text{blocked } \in \text{Seq(active)} \\
\end{aligned}
\]
A simple scheduler (ctd)

\[
\text{Init} \triangleq \\
\land \text{active} = \{\} \\
\land \text{procs} = \langle\rangle \\
\land \text{running} = \text{IDLE} \\
\land \text{ready} = \langle\rangle \\
\land \text{blocked} = \langle\rangle \\
\]

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

\[
\text{Run}(p) \triangleq \\
\land \text{running} = \text{IDLE} \\
\land \text{ready} \neq \langle\rangle \\
\land p = \text{Head(ready)} \\
\land \text{running}' = p \\
\land \text{procs}' = [\text{procs EXCEPT } ![p] = [\text{priority} \mapsto \text{procs}[p].\text{priority}, \text{state} \mapsto \text{"running"}]] \\
\land \text{ready}' = \text{Tail(ready)} \\
\land \text{UNCHANGED } \langle \text{active, blocked}\rangle \\
\]
A simple scheduler (ctd)

\[ Yield(p) \triangleq \]
\[ \land running = p \]
\[ \land running' = IDLE \]
\[ \land procs' = [procs EXCEPT ![p] = [priority \mapsto procs[p].priority, state \mapsto “ready”]] \]
\[ \land ready' = Append(ready, p) \]
\[ \land UNCHANGED \langle active, blocked \rangle \]

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

\[ Next \triangleq \]
\[ \exists p \in PROCID : \]
\[ \lor Create(p) \]
\[ \lor Run(p) \]
\[ \lor Yield(p) \]
\[ \lor Terminate(p) \]

\[ Range(f) \triangleq \{f[x] : x \in \text{\textsc{domain}} f\} \]
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.

```plaintext
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.
either The either statement has the form:

```plaintext
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 ∈ 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 ∈ {expr} do ...`
\* BEGIN TRANSLATION
VARIABLES big, small

text continues...

\* END TRANSLATION
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.

```plaintext
--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 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

```plaintext
/* 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, $s_1$, $s_2$, and $s_3$, 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 \( \text{Done} \).

We can define a \( \text{BalanceOk} \) property as follows:

\[
\text{BalanceOk} \equiv (pc[1] = "Done" \lor pc[2] = "Done") \implies \text{balance} = 80
\]

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.