Embedded Systems Specification and Design
Model-based Design and Verification

David Kendall

Introduction

- Functions
- CHOOSE
- IF . . . THEN . . . ELSE
- Tuples and sequences
- Strings
- Illustrative example — a simple scheduler

A fancier building control system

Informal statement of requirements:

- The system should monitor persons as they enter and leave buildings
- The system should maintain a register of known persons. It should be possible to remove a person from the register
- The system should maintain information about *access permissions*, i.e. which buildings any registered person is allowed to enter. It should be possible both to grant and remove access permissions
- A person can only enter a building if they are registered and have permission for entry to the building
- The system should keep track of the location of persons on the register

We'll use this as a running example to introduce some more of the TLA$^+$ language

Representing the state of the system

The informal specification mentions

- *persons* who should be monitored
- *buildings* that persons may enter and leave
- a *register* of users
- information about *access permissions* of registered users
- information about *location* of registered users

CONSTANT

PERSON,	set of persons
BUILDING	set of buildings

VARIABLE

register	set of registered users
permission	access permissions of registered users
location	location of registered users

How to represent the access permissions?

We have declared a variable, *permission*, to represent the access permissions of registered users

What kind of values should *permission* be able to take?

Suppose

$$\textit{PERSON} = \{\text{"p1"}, \text{"p2"}, \text{"p3"}, \text{"p4"}\} \quad \textit{register} = \{\text{"p1"}, \text{"p4"}\}$$
$$\textit{BUILDING} = \{\text{"b1"}, \text{"b2"}, \text{"b3"}\}$$

give some examples of what *permission* might look like

These are written in TLA$^+$ as functions

What is a function in TLA$^+$?

Programmers can think of functions as being like arrays, i.e. a mapping from some index set to some set of values

Arrays in a programming language are often restricted to a very limited index set, e.g. in C an array of size N has an index set that is the set of natural numbers from 0 to $N - 1$

In TLA$^+$ functions are primitive objects whose index set can be any set, even an infinite set, e.g.

$$f \quad \widehat{=} \quad [n \in 1..5 \mapsto n + 1]$$

defines a function, f, that maps every number n in the set $\{1, 2, 3, 4, 5\}$ to its successor $n + 1$

You can read $\mapsto$ as “maps to”. It is written as $|->$ in ASCII.

TLA$^+$ functions (ctd)

We can define a function *sqr*

$$sqr \quad \widehat{=} \quad [i \in Nat \mapsto i^2]$$

where *Nat* is the infinite set of natural numbers

TLA$^+$ provides an alternative syntax for this kind of function definition, e.g.

$$sqr2[i \in Nat] \quad \widehat{=} \quad i^2$$

Function application is written using square brackets, e.g. $sqr[3]$, which is equal to 9

A function is only defined for elements in its index set. In TLA$^+$ the index set of a function is called its DOMAIN , e.g. DOMAIN *f* is $\{1, 2, 3, 4, 5\}$ and DOMAIN *sqr* is *Nat*

TLA$^+$ functions (ctd)

Two functions f and g are equal iff they have the same domain and $f[x] = g[x]$ for all x in their domain, e.g. $sqr = sqr2$

The *range* of a function f is just the set of values $f[x]$ for all x in the domain of f. It can be defined as follows in TLA$^+$

$$Range(f) \quad \widehat{=} \quad \{f[x] : x \in \text{DOMAIN } f\}$$

$[\mathbf{S} \to \mathbf{T}]$
This expression denotes the set of all functions f whose domain is S and whose range is any subset of T

Exercise Let $A = 1..3$ and $B = \{\text{"a"}, \text{"b"}\}$. Calculate the set $[B \to A]$. Use TLC to check your answer.

The empty function, whose domain is the empty set, is written as $\langle\rangle$ (<< >> in ASCII)

Revisiting the state of the system

CONSTANT
PERSON,
BUILDING

$OUTSIDE \triangleq$ CHOOSE $x : x \notin PERSON \cup BUILDING$

VARIABLE
register
permission
location

How should we define *TypeOk*?

$TypeOk \triangleq$
$\land\ register \subseteq PERSON$
$\land\ permission \in [register \to \text{SUBSET } BUILDING]$
$\land\ location \in [register \to BUILDING] \land location \in [register \to (BU$

The CHOOSE operator

The CHOOSE operator is related to the existential quantifier $\exists$

The formula $\exists\, x \in S : P(x)$ asserts that there is a value x for which $P(x)$ is TRUE. If that is the case, then CHOOSE $x \in S : P(x)$ equals such a value

We can use CHOOSE to select the unique value that satisfies the specified property

Example: We can define the maximum of a set of integers like this

$$Max(S) \mathrel{\widehat{=}} \text{CHOOSE } x \in S : \boxed{\forall\, y \in S : x \geq y}$$

CHOOSE satisfies the following property

$$\begin{aligned} \exists\, x \in S : P(x) \Rightarrow \quad & \wedge\, (\text{CHOOSE } x \in S : P(x)) \in S \\ & \wedge\, P(\text{CHOOSE } x \in S : P(x)) \end{aligned}$$

If there is no element x in S satisfying $P(x)$, then we don't know anything about the value CHOOSE $x \in S : P(x)$

The Enter operation

Assume $p \in PERSON$ and $b \in BUILDING$

How can we model an action which changes the state so that person p enters building b?

$$
\begin{aligned}
&Enter(p, b) \mathrel{\widehat{=}} \\
&\quad \land\ p \in register \\
&\quad \land\ b \in permission[p] \\
&\quad \land\ location'[p] = b \qquad ???? \qquad \land\ location' = [location \text{ EXCEPT } ![p \\
&\quad \land\ register' = register \land permission' = permission \quad \land \text{ UNCHAN}
\end{aligned}
$$

The EXCEPT construct

Programmers may think that it should be possible to update *location* in the *Enter* operation by writing

$$location'[p] = b$$

Why is this wrong?

Because it only specifies the new value of $location[p]$. It says nothing about the value of any of the other elements in the domain of the *location* function

It doesn't even require that the new *location* function have the same domain

We could specify the new value completely like this

$$\begin{aligned} location' = \quad & [x \in \text{DOMAIN } location \mapsto \\ & \quad \text{IF } x = p \text{ THEN } b \text{ ELSE } location[x]\,] \end{aligned}$$

TLA$^+$ allows us to shorten this long-winded formulation using EXCEPT,

$$location' = [location \text{ EXCEPT } ![p] = b]$$

IF...THEN...ELSE

The value of IF p THEN e_1 ELSE e_2 is e_1 if p is *TRUE* and e_2 if p is *FALSE*

IF expressions can be nested, e.g.

$$\text{IF } i > 0 \text{ THEN } 1 \text{ ELSE } (\text{IF } i = 0 \text{ THEN } 0 \text{ ELSE } -1)$$

or

$$\text{IF } i \geq 0 \text{ THEN } (\text{IF } i = 0 \text{ THEN } 0 \text{ ELSE } 1) \text{ ELSE } -1$$

The Leave operation

Exercise: Take to 2 to 3 minutes to write down a specification of the *Leave* operation, by which it is recorded that person p leaves building b

$$
\begin{array}{l}
Leave(p, b) \mathrel{\widehat{=}} \\
\quad \land\ p \in register \\
\quad \land\ location[p] = b \\
\quad \land\ location' = [location \text{ EXCEPT } ![p] = OUTSIDE] \\
\quad \land\ \text{UNCHANGED } \langle register, permission \rangle
\end{array}
$$

The Register operation

$$
\begin{array}{l}
\mathit{Register}(p) \mathrel{\widehat{=}} \\
\quad \land\ p \in \mathit{PERSON} \setminus \mathit{register} \\
\quad \land\ \mathit{register}' = \mathit{register} \cup \{p\} \\
\quad \land\ \mathit{permission}' = [x \in \mathrm{DOMAIN}\ \mathit{permission} \cup \{p\} \mapsto \\
\qquad\qquad \mathrm{IF}\ x \in \mathrm{DOMAIN}\ \mathit{permission} \\
\qquad\qquad \mathrm{THEN}\ \mathit{permission}[x] \\
\qquad\qquad \mathrm{ELSE}\ \{\}] \\
\quad \land\ \mathit{location}' = [x \in \mathrm{DOMAIN}\ \mathit{location} \cup \{p\} \mapsto \\
\qquad\qquad \mathrm{IF}\ x \in \mathrm{DOMAIN}\ \mathit{location} \\
\qquad\qquad \mathrm{THEN}\ \mathit{location}[x] \\
\qquad\qquad \mathrm{ELSE}\ \mathit{OUTSIDE}]
\end{array}
$$

In this case, the *permission* and *location* functions need to be extended to include p in their domains. This prevents us from using a formulation with EXCEPT and we must use the more verbose form to ensure that the domains of the new functions are specified and mappings determined for every element

The Next operation

$$
\begin{array}{l}
\mathit{Next} \mathrel{\widehat{=}} \\
\quad \exists\, p \in \mathit{PERSON}, b \in \mathit{BUILDING} : \\
\qquad \lor\ \mathit{Register}(p) \\
\qquad \lor\ \mathit{DeRegister}(p) \\
\qquad \lor\ \mathit{AddPermission}(p, b) \\
\qquad \lor\ \mathit{RevokePermission}(p, b) \\
\qquad \lor\ \mathit{Enter}(p, b) \\
\qquad \lor\ \mathit{Leave}(p, b)
\end{array}
$$

The *AddPermission*, *RevokePermission*, and *DeRegister* operations are left as exercises

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 ith 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

$Seq(S)$	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:

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

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

Instead of writing $p[\text{"name"}]$ or $p[\text{"age"}]$, 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

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

A simple scheduler

MODULE *SimpleScheduler*

EXTENDS *Naturals*, *Sequences*

CONSTANT
PROCID,
PRIORITY

VARIABLE
active,
procs,
running,
ready,
blocked

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

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

$TypeOk \triangleq$
$\land\ active \subseteq PROCID$
$\land\ procs \in [active \to Process]$
$\land\ running \in active \cup \{IDLE\}$
$\land\ ready \in Seq(active)$
$\land\ blocked \in Seq(active)$

A simple scheduler (ctd)

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

A simple scheduler (ctd)

$$
\begin{array}{l}
Yield(p) \triangleq \\
\quad \land \ running = p \\
\quad \land \ running' = IDLE \\
\quad \land \ procs' = [procs \ \text{EXCEPT} \ ![p] = [priority \mapsto procs[p].priority, \ state \mapsto \text{“ready”}]] \\
\quad \land \ ready' = Append(ready, \ p) \\
\quad \land \ \text{UNCHANGED} \ \langle active, \ blocked \rangle \\
\\
Terminate(p) \triangleq \\
\quad \land \ running = p \\
\quad \land \ running' = IDLE \\
\quad \land \ active' = active \setminus \{p\} \\
\quad \land \ procs' = [pp \in \text{DOMAIN} \ procs \setminus \{p\} \mapsto procs[pp]] \\
\quad \land \ \text{UNCHANGED} \ \langle ready, \ blocked \rangle \\
\\
Next \triangleq \\
\quad \exists \, p \in PROCID : \\
\qquad \lor \ Create(p) \\
\qquad \lor \ Run(p) \\
\qquad \lor \ Yield(p) \\
\qquad \lor \ Terminate(p) \\
\\
Range(f) \triangleq \{f[x] : x \in \text{DOMAIN} \ f\}
\end{array}
$$