Tasking Event-B Translations

A. Edmunds

July 13, 2010

1 Tasking Event-B Machines to IL1

Source	Target
Tasking Machine	Task (DataType)
AutoTask Machine	Task (non-DataType / only run on start-up)
Shared Machine	Protected Object (DataType)

Tasking and shared machine events map to Subroutine declarations.

Each reference to an event in the control construct is either 'local' or 'remote' with respect to a task. Remote events, used in looping and branching constructs, have no guards (or they have trivially true guards). Local events used in the $EventWrapper$ construct have no guards.

2 Synchronized Local/Remote Events

In order to represent the combined updates on local and remote machines in the tasking language we introduce synchronized event composition, for events of tasking/shared machines. In the following discussion we compose one local and one remote event e_l and e_r respectively using $\|_e$; where the remote event does not have any guards. We define the combined event e as

$$e \triangleq e_l \|_e e_r \tag{1}$$

In version1 of the tasking language local and remote machines do not share state, the variables of the guards and actions are disjoint. We can write the individual assignments of $e \triangleq e_l \|_e e_r$ in the following way, e_l, $x_1, .., x_j := y_1 .. y_j$ and the assignments of e_r, $x_{j+1}, .., x_n := y_{j+1} .. y_n$. When composed in parallel we have,

$$e_l \|_e e_r \triangleq g_l \rightarrow x_1, .., x_n := y_1 .. y_n \tag{2}$$

When events are used in an $EventWrapper$ construct, the implementation maps to a blocking call. In this case it makes no sense to have a guard on the local

event since the calling task should not block itself. So we only guard the remote event. We define the compound event e as,

$$e = a_l \parallel_e g_r \rightarrow a_r \tag{3}$$

When we use $IfThenElse$ or $While$ constructs, we restrict the use of guards to the local event only. We prohibit the use of guards in remote events to avoid complications due to interleaving with other tasks. Our previous work, with OCB, had a similar constraint for the same reason; and the restriction also allows the developer to reason about the effects in a clear way (problem with false *else* guards!!). This also means that it makes no sense to write a branch without a guarded local event, since the remote event has no guards, **if**($true$) **then** a **endif** is simply equivalent to the update a.
A compound event e is defined as,

$$e = g_l \rightarrow a_l \parallel_e a_r \tag{4}$$

If one of the events, either local or remote, is not specified in the control construct then the missing event is interpreted as,

$$\top \rightarrow skip \tag{5}$$

2.1 Tasking - Parallel Events to IL1

Control	IL1
$Control1$; $Control2$	$Control1$; $Control2$
`EventWrapper` e $e = e_l \parallel_e e_r$	`call` e_l`();` `call` $target.e_r$`()`

L`: DO` e `OD`
$e = g_l \rightarrow a_l \parallel_e e_r$
	In task body:
	`while(`g_l`){`
	` call` L_{er}`( );` a_l`;`
	`}`
	In Protected:
	`subroutine` L_{er}`( ){` a_r`; }`

L`: DO` e_1 `FINALLY` e_2 `OD`
$e_x = g_{xl} \rightarrow a_{xl} \parallel_e e_{xr}$
	In task body:
	`while(` g_{1l} `){`
	` call` L_{e1r}`();` a_{1l}`;`
	`}`
	`call` e_{2r}`();` a_{2l}`;`
	
	In Protected:
	`subroutine` L_{e1r}`( ){` a_{1r}`; }`

Control	IL1

L: `IF` e_1 `ENDIF`
`[ ELSEIF` e_x `ENDELSEIF]`..
`[ ELSE` e_n `ENDELSE]`
	In task body:
	`[if(` g_{1l} `){` *body* `}]`
	`[elseif(`g_{xl}`){` *body* `}]`..
	`[else{` *body* `}]`

x$\in 1\,..\,n$ $e_x = g_{xl} \rightarrow a_{xl} \parallel_e e_{xr}$	
	body $\triangleq$ `call` L_{exr}`();` a_{xl} and in Protected: `subroutine` L_{exr}`(){` a_{xr} `}`

2.2 Tasking - Parallel Events to Event-B

2.2.1 Using Labelled Clauses

In the following table we use e_l to indicate an event that is local to a task, and e_r to indicate a (remote) event belonging to a shared machine.

Control	Event-B
$Control1$; $Control2$	$Control1$; $Control2$
L: `EventWrapper` e $e \triangleq a_l \parallel_e g_r \rightarrow a_r$	$e_l \triangleq$ **WHEN** $pc_t = L$ **THEN** $a_l \parallel pc_t := next(L)$ **END** $e_r \triangleq$ **WHEN** g_r **THEN** a_r **END**
L: `DO` e `OD` $e = g_l \rightarrow a_l \parallel_e a_r$	$e_{lwhile} \triangleq$ **WHEN** $g_l \wedge pc_t = L$ **THEN** a_l **END** $e_{rwhile} \triangleq$ **WHEN** $\top$ **THEN** a_r **END** $e_{lwhilefalse} \triangleq$ **WHEN** $\neg g_l \wedge pc_t = L$ **THEN** $pc_t := next(L)$ **END**
L: `DO` e_1 `FINALLY` e_2 `OD` $e_x = g_{xl} \rightarrow a_{xl} \parallel_e a_{xr}$	$e_{lwhile} \triangleq$ **WHEN** $g_{1l} \wedge pc_t = L$ **THEN** a_{1l} **END** $e_{rwhile} \triangleq$ **WHEN** $\top$ **THEN** a_{1r} **END** $e_{lfinally} \triangleq$ **WHEN** $\neg g_{1l} \wedge g_{2l} \wedge pc_t = L$ **THEN** $a_{2l} \parallel pc_t := next(L)$ **END** $e_{rfinally} \triangleq$ **WHEN** $\top$ **THEN** a_{2r} **END**

Control	Event-B
L: `IF` e_1 `ENDIF` `[ ELSEIF` e_x `ENDELSEIF]`.. `[ ELSE` e_n `ENDELSE ]` $e_x = g_{xl} \rightarrow a_{xl} \parallel_e a_{xr}$	$e_{lif} \triangleq$ **WHEN** $g_{1l} \ \wedge \ pc_t = L$ **THEN** $a_{1l} \parallel \ pc_t := next(L)$ **END** $e_{rif} \triangleq$ **WHEN** $\top$ **THEN** a_{1r} **END** $e_{lelseif\_x} \triangleq$ **WHEN** $\bigwedge \neg g_{l1..(x-1)} \wedge \ g_{lx} \ \wedge \ pc_t = L$ **THEN** $a_{lx} \parallel \ pc_t := next(L)$ **END** $e_{relseif\_x} \triangleq$ **WHEN** $\top$ **THEN** a_{rx} **END** $e_{lelse\_x} \triangleq$ **WHEN** $\bigwedge \neg g_{1..(n-1)l} \ \wedge \ pc_t = L$ **THEN** $a_{nl} \parallel \ pc_t := next(L)$ **END** $e_{relse\_x} \triangleq$ **WHEN** $\top$ **THEN** a_{nr} **END**

2.2.2 Without Labelled Clauses

In the following table we use e_l to indicate an event that is local to a task, and e_r to indicate a (remote) event belonging to a shared machine.

Control	Event-B
$Control1$; $Control2$	$Control1$; $Control2$
`EventWrapper` e $e \triangleq a_l \parallel_e g_r \rightarrow a_r$	$en_l \triangleq$ **WHEN** $en = TRUE$ **THEN** $a_l \parallel en := FALSE \parallel$ $next(en) := TRUE$ **END** $e_r \triangleq$ **WHEN** g_r **THEN** a_r **END**
`DO` e `OD` $e_x = g_l \rightarrow a_l \parallel_e a_r$	$e_{lwhile} \triangleq$ **WHEN** $g_l \wedge en = TRUE$ **THEN** a_l **END** $e_{rwhile} \triangleq$ **WHEN** $\top$ **THEN** a_r **END** $e_{lfinally} \triangleq$ **WHEN** $\neg g_l \wedge en = TRUE$ **THEN** $en := FALSE \parallel$ $next(en) := TRUE$ **END**
`DO` e_1 `FINALLY` e_2 `OD` $e_x = g_{xl} \rightarrow a_{xl} \parallel_e a_{xr}$	$e_{lwhile} \triangleq$ **WHEN** $g_{1l} \wedge en = TRUE$ **THEN** a_{1l} **END** $e_{rwhile} \triangleq$ **WHEN** $\top$ **THEN** a_{1r} **END** $e_{lfinally} \triangleq$ **WHEN** $\neg g_{1l} \wedge en = TRUE$ **THEN** $a_{2l} \parallel en := FALSE \parallel$ $next(en) := TRUE$ **END** $e_{rfinally} \triangleq$ **WHEN** $\top$ **THEN** a_{2r} **END**

Control	Event-B
L: `IF` e_1 `ENDIF` `[ ELSEIF` e_x `ENDELSEIF]`.. `[ ELSE` e_n `ENDELSE ]`	$e_{lif} \triangleq$ **WHEN** $g_{1l} \ \wedge \ en = TRUE$ **THEN** $a_{1l} \parallel \ en := FALSE$ $next(en) := TRUE$ **END**
$e_x = g_{xl} \rightarrow a_{xl} \parallel_e a_{xr}$	$e_{rif} \triangleq$ **WHEN** $\top$ **THEN** a_{1r} **END**
	$e_{lelseif\_x} \triangleq$ **WHEN** $\bigwedge \neg g_{l1..(x-1)} \wedge \ g_{lx} \ \wedge$ $en = TRUE$ **THEN** $a_{lx} \parallel \ next(en) := TRUE \parallel$ $en := FALSE$ **END**
	$e_{relseif\_x} \triangleq$ **WHEN** $\top$ **THEN** a_{rx} **END**
	$e_{lelse\_x} \triangleq$ **WHEN** $\bigwedge \neg g_{1..(n-1)l} \ \wedge$ $en = TRUE$ **THEN** $a_{nl} \parallel \ next(en) := TRUE \parallel$ $en := FALSE$ **END**
	$e_{relse\_x} \triangleq$ **WHEN** $\top$ **THEN** a_{nr} **END**