Tasking Event-B Translations

A. Edmunds

July 20, 2011

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)
Environ Machine	Model of the Environment

In our approach it is usually the case that the Events of Tasking and Shared machines map to Subroutine declarations, exceptions to this will be described at the appropriate time. Tasks have a *TaskBody* construct in which the task behaviour is defined; *EventWrappers* may appear in the task body to contain Event synchronizations. In a synchronization (*SynchEvents*) an event is either 'local' or 'remote' with respect to a task. There are restrictions on which events may have guards. Remote events, used in looping and branching constructs, should not be guarded. Local events, used in the *EventWrapper* construct, should also not be guarded. Synchronization occurs between a pair of events; one from a *Tasking Machine*, and one from a *Shared Machine* or *Environ Machine*. No synchronization occurs between *Tasking Machines*. The abstract syntax follows:

```
TaskBody ::= Seq | Branch | Do | EventWrapper | Output

Seq  ::=  TaskBody TaskBody

Branch ::=  Body [SubBranch] Else

SubBranch ::=  Body [SubBranch]

Else ::=  EventWrapper

Body ::=  EventWrapper

Do ::=  Body [Finally]
```

```
Finally ::=  EventWrapper

Output ::=  Text Variable

EventWrapper ::=  SynchEvents

SynchEvents ::=  Local Remote

Local ::=  Event

Remote ::=  Event
```

2 Synchronized Local/Remote Events

To represent the combined updates on local and remote machines we introduce synchronized event composition. The synchronization of the two events is equivalent to a single atomic event, with the guards and actions of the individual events merged. We can write the guards and actions of the events as guarded commands [1]. The general case of event synchronization is shown in Equation (1) where a local event e_l is written as $g_l \rightarrow a_l$, and g_l and a_l are local guards and actions. The remote event e_r is written as $g_r \rightarrow a_r$, where g_r and a_r are remote guards and actions. The synchronization of one local and one remote event uses the event composition operator $\|_e$. The actions describing state updates are composed with the parallel update operator $\|$.

$$g_l \rightarrow a_l \|_e g_r \rightarrow a_r \triangleq g_l \wedge g_r \rightarrow a_l \| a_r \tag{1}$$

So we can aslo write the combined event e_c as

$$e_c \triangleq e_l \|_e e_r \tag{2}$$

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 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,

$$e_c \triangleq g_l \wedge g_r \rightarrow x_1, .., x_n := y_1 .. y_n \tag{3}$$

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 restrict the guarded compound event e_g as follows,

$$e_g = a_l \|_e g_r \rightarrow a_r \tag{4}$$

When we use *branch* or *loop* 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 branching event e_b and looping event e_w is restricted as follows,

$$e_b = e_w = g_l \rightarrow a_l \parallel_e a_r \tag{5}$$

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

$$\top \rightarrow skip \tag{6}$$

2.1 Tasking - Parallel Events to IL1

We describe here the mapping between Tasking Event-B and the IL1 meta-model. The $Control$ constructs populate the $TaskBody$ of the source $Tasking\ Machine$, and a similarly named construct is used to populate the target $TaskBody$.

Control	$< Control >^T$
$Control1$; $Control2$	$< Control1 >^T$; $< Control2 > T$

`EventWrapper` e_s
where $e_s = e_l \parallel_e e_r$
$e_l = a_l$, and $e_r = g_r \rightarrow a_r$
	`call` e_l`();` `call` $target.e_r$`()`
	In the task: `subroutine` e_l`(){` a_l`; }`
	In Protected:
	`subroutine` e_r`()when(`g_r`){` a_r`; }`

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

L: `DO` e_{1w} `FINALLY` e_{2w} `OD`
$e_{iw} = g_{il} \rightarrow a_{il} \parallel_e e_{ir}$
where $i \in 1\,..\,2$
	In task body:
	`while(` g_{1l} `){`
	`call` e_{1r}`();` a_{1l}`;`
	`}`
	`call` e_{2r}`();` a_{2l}`;`
	
	In Protected:
	`subroutine` e_{1r}`( ){` a_{1r}`; }`
	`subroutine` e_{2r}`( ){` a_{2r}`; }`

Control	IL1
	In task body:
L: `IF` e_{1b} `ENDIF`	`[if(` g_{1l} `){` *body* `}]`
`[ ELSEIF` e_{ib} `ENDELSEIF]`..	`[elseif(`g_{il}`){` *body* `}]`..
`[ ELSE` e_{nb} `ENDELSE]`	`[else{` *body* `}]`
$i \in 1 \mathinner{..} n$	
$e_{ib} = g_{il} \to a_{il} \parallel_e e_{ir}$	
	body $\triangleq$
	`call` e_{ir}`();` a_{il}
	and in Protected:
	`subroutine` e_{ir}`(){` a_{ir} `}`

2.2 Tasking - Parallel Events to Event-B

2.2.1 Using Labelled Clauses

Control	Event-B
$Control1$; $Control2$	
L: `EventWrapper` e_c $e_c \triangleq a_l \parallel_e g_r \rightarrow a_r$ where $next(L)$ is a function returning the next program counter label.	$e_l \triangleq$ **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_w `OD` $e_w = g_l \rightarrow a_l \parallel_e a_r$ where $next(L)$ is a function returning the next program counter label.	$e_{lwhile} \triangleq$ **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_{1w} `FINALLY` e_{2w} `OD` where $i \in 1 .. 2$ $e_i = g_{il} \rightarrow a_{il} \parallel_e a_{ir}$ where $next(L)$ is a function returning the next program counter label.	$e_{lwhile} \triangleq$ **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_{1b} `ENDIF`
`[ ELSEIF` e_{ib} `ENDELSEIF]..`
`[ ELSE` e_{nb} `ENDELSE ]`
where $i \in 1 \,..\, n$
$e_{ib} = g_{il} \rightarrow a_{il} \parallel_e a_{ir}$
	$e_{lif} \triangleq$
	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

where $next(L)$ is a function returning the next program counter label.	$e_{lelseif\_i} \triangleq$ **WHEN** $\bigwedge \neg g_{l1..(i-1)} \wedge\; g_{il} \;\wedge\; pc_t = L$ **THEN** $a_{il} \parallel\; pc_t := next(L)$ **END** $e_{relseif\_i} \triangleq$ **WHEN** $\top$ **THEN** a_{ir} **END** $e_{lelse\_i} \triangleq$ **WHEN** $\bigwedge \neg g_{1..(i-1)l} \;\wedge\; pc_t = L$ **THEN** $a_{nl} \parallel\; pc_t := next(L)$ **END** $e_{relse\_i} \triangleq$ **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 $e \triangleq a_l \parallel_e g_r \rightarrow a_r$, and en is a representation of the program counter derived from the event name. Where next(en) is a function returning the next enabled counter.	$e_l \triangleq$ **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` $e = g_l \rightarrow a_l \parallel_e a_r$, and en is a representation of the program counter derived from the event name. Where next(en) is a function returning the next enabled counter.	$e_{lwhile} \triangleq$ **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` $i \in 1..2$ $e_i = g_{il} \rightarrow a_{il} \parallel_e a_{ir}$, and en is a representation of the program counter derived from the event name. Where next(en) is a function returning the next enabled counter.	$e_{lwhile} \triangleq$ **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_i `ENDELSEIF]..`
`[ ELSE` e_n `ENDELSE ]`
 	
$i \in 1..n$
$e_i = g_{il} \to a_{il} \parallel_e a_{ir}$,
and *en* is a representation of the program counter derived from the event name.
 	
Where next(en) is a function returning the next enabled counter.
	$e_{lif} \triangleq$
	WHEN $g_{1l} \ \wedge \ en = TRUE$
	THEN $a_{1l} \parallel \ en := FALSE$
	$next(en) := TRUE$
	END
	$e_{rif} \triangleq$
	WHEN $\top$
	THEN a_{1r}
	END
	$e_{lelseif\_i} \triangleq$
	WHEN $\bigwedge \neg g_{l1..(i-1)} \wedge \ g_{il} \wedge$
	$en = TRUE$
	THEN $a_{il} \parallel \ next(en) := TRUE \parallel$
	$en := FALSE$
	END
	$e_{relseif\_i} \triangleq$
	WHEN $\top$
	THEN a_{ir}
	END
	$e_{lelse\_i} \triangleq$
	WHEN $\bigwedge \neg g_{1..(n-1)l} \ \wedge$
	$en = TRUE$
	THEN $a_{nl} \parallel \ next(en) := TRUE \parallel$
	$en := FALSE$
	END
	$e_{relse\_i} \triangleq$
	WHEN $\top$
	THEN a_{nr}
	END

3 Synchronizing Events using the ProcedureSynch Construct

In a ProcedureSynch we have local and remote events e_l and e_r respectively. We have a local event with an ordered set of parameters P and variables V. A local event can synchronize with a remote event with ordered sets of Parameters Q and variables W. The synchronization e_s can be written in the form of the Guarded Command,

$$e_s \triangleq g_l(P, V) \to a_l(P, V) \parallel_e g_r(Q, W) \to a_r(Q, W)$$

In the translation to the Common Language Model we find the guards and actions play a roll in the direction of parameter passing see Table 1. In the table we identify individual Parameters $p \in P$ and $q \in Q$, and individual Variables

Event-B	Direction	Type
parameter p, where $p = v$	out	actual
parameter p where $v := p$	in	actual
parameter q, where $q = w$	out	formal
parameter q, where $w := q$	in	formal

Table 1: Parameters: Type and Direction from use in Guards and Actions

$v \in V$ and $w \in W$. Remembering subscript l represents a task local event's guards and actions, and subscript r represents the remote event guards and actions, occurring in a shared machine. As a general observation, outgoing parameters appear only in event guards, and incoming parameters appear only on the RHS of assignment actions.

4 Sensing and Actuation Events

We introduce a type of machine called an Environ Machine which models the environment. The tasks of a development interact with the environment by reading monitored variables (sensing) and setting controlled variables (actuation). The monitored and controlled variables reside in the environment.

There are two approaches that we use to facilitate communication with the environment. In the first case we use Memory Mapped IO using Addressed Variables. We relate event parameters of sensing/actuating tasks to memory locations, Addressed Variables consist of a base b and location loc. This approach is suitable for deployable code, and if we relate the monitored and controlled variables of the environment to these memory locations we are also able to simulate the environment using these addresses. We may also choose to ignore Memory Mapped IO (the second case) and simply interact with the environment using subroutine calls. In Ada, these are implemented using entry calls, from the task to the environment. This approach may also be used to simulate the environment, for instance, if the developer does not have the addresses available; it may also be used to simulate calls to a device driver API. This approach gives rise to the options summarized in the following table:

Generated Code	Tasking Development
Tasks write to specific memory addressed variables for deployment.	Extend task event parameters with address variables.
Tasks and the Environment simulation interact through memory addressed variables.	Extend task event parameters and environ machine variables with address variables.
Tasks and Environment simulation interact using subroutine calls	Use no address variables

In Tables 2, 3 and 4 we describe the translation between the tasking annotations and the generated code. In the tables, the local event has parameters p

Tasking Event-B	Translation
Actuating Local Evt: , with Parameter addr(b,loc) p, Var v, and guard $p = v$	Add Addressed Variable p to task, with base b and location loc. Use $p := v$ in s.
Actuating Remote Evt: with parameter q, Var w, and action $w := q$.	Ignore environment.
Sensing Local Evt: with Parameter addr(b,loc) p, Var v, and action $v := p$	Add Addressed Variable p to task, with base b and location loc. Use $v := p$ in s.
Sensing Remote Evt: with Parameter q, Var w, and guard $q = w$	Ignore environment.

Table 2: Deployment: Translation of Sensing/Actuation Parameters

Tasking Event-B	Translation
Actuating Local Evt: with Parameter p, Var v, and guard $p = v$.	Map v to an actualOut parameter.
Actuating Remote Evt: with Parameter q, Var w, and action $w := q$.	Map q to a formalIn parameter. Use $w := q$ in s.
Sensing Local Evt: with parameter p, Var v, and action $v := p$.	Map v to an actualIn parameter.
Sensing Remote Evt: with parameter q, Var w, and guard $q = w$.	Map q to a formalOut parameter. Use $q := w$ in s.

Table 3: Simulation1: Sensing/Actuating with Subroutine Calls

and sensed/actuated variables v. A local event can synchronize with a remote event, with parameters q , and monitored/controlled variables w. It can be seen in the tables that an event, in a machine, usually maps to a subroutine s; but is sometimes ignored since it is used for reference only. In the Tasking Event-B approach we stipulate that parameter ordering is critical; we match parameters of the local and remote events when we translate to the subroutine declaration (signature) and call parameters. The subroutine signature contains the formal parameters, and the subroutine call contains the actual parameters. Events that interact with the environment are marked as either Sensing or Actuating, so that the translator can take appropriate action.

Tasking Event-B	**Translation**
Actuating Local Evt: with Parameter addr(b,loc) p, Var v and guard $p = v$.	Add Addressed Variable p to the task, with base b and location loc. Use $p := v$ in s.
Actuating Remote Evt: with parameter addr(b,loc) q, Var w and action $w := q$.	Add Addressed Variable w to Environ task, with base b and location loc. Ignore event, i.e. no translation to s.
Sensing Local Evt: with parameter addr(b,loc) p, Var v, and action $v := p$.	Add Addressed Variable p to task, with base b and location loc. Use $v := p$ in s.
Sensing Remote Evt: with parameter addr(b,loc) q, Var w, and guard $q = w$.	Add Addressed Variable w to Environ task, with base b and location loc. Ignore event, i.e. no translation to s.

Table 4: Environ Simulation2: Sensing/Actuating with Addressed Variables

4.1 The role of Synchronisation in Sensing and Actuation

We can write the specification of a sensing event synchronization in the style of the Guarded Command Language, as follows:

$$g_l(P,V) \rightarrow a_l(P,V) \parallel_e g_r(Q,W) \rightarrow a_r(Q,W)$$

We know that sensing/actuating synchronizations are atomic, and we map a synchronization to a single subroutine in which either the environment variables are read or updated. We do not allow sensing and actuating in the same event. We use the following notation: Tasks have ordered sets of Parameters P and Variables V. The Environ machine has ordered sets of Parameters Q and Variables W. Variables V are the sensed/actuated variables, Variables W are the monitored/controlled variables. Event Parameters are paired (using order of declaration) between the synchronized machines. When we write $P = Q$, we mean that we have two ordered sets where n elements of P are paired with the n elements of Q, such that, for each $i \in 0..n-1$, we have $P_i = Q_i$. Therefore we can substitute each element P_i for its paired value Q_i and vice versa. $V := P$ is the simultaneous substitution of the paired elements in V and P.

In a task sensing event e_l we have,

$$V := P$$

Now, in the corresponding synchronised event e_r, from the guard, we have $W = Q$. We know that $P = Q$ from paired parameters, so $P = W$. Therefore,

$$V := W$$

This means that the monitored variable values W are assigned to the sensed variables V.

In a task actuating event e_l we have,

$$V = P$$

and in the environment event e_r,

$$W := Q$$

We know that $P = Q$ from paired parameters, so $Q = V$, and therefore,

$$W := V$$

This means that the values of the actuated variables V are assigned to the controlled variables W.

4.2 Implementing Memory Mapped IO

Memory Mapped IO is specified in Tasking Event-B using Addressed Variables. Addressed Variables provide a way of specifying a memory location; whenever the variable is used in an expression, the value is retrieved from the specified memory location. Local event e_l has an ordered set of Parameters P which are mapped to an ordered set of memory locations M. When we write $P = M$ we mean that we have two ordered sets, where n elements of P are paired with the n elements of M. For each $i \in 0 \mathinner{..} n-1$, we have $P_i = M_i$. Therefore we can substitute each element P_i for its paired value M_i. Variables W of the Environ Machine are also paired with the corresponding memory locations M in the same way. We write $W = M$ and may substitute each W_i for M_i and vice versa. We use the guards to relate the two as follows in the local sensing event e_l,

$$V := P$$

From the address mapping we have $P = M$, so,

$$V := M$$

The values from M are assigned to the variable V.

That is all we are interested in for deployable code, but in a simulation using Addressed Variables in the environment we additionally wish to show that the memory locations M are updated. In an environment model we specify events to manipulate the environment simulation. In an environment simulation some variable W are updated using the following expression,

$$W := E(W, Q)$$

and from the address mapping we have $W = M$ so,

$$M := E(W, Q)$$

These are the memory values M, read by the local sensing event e_l when performing the update $V := M$.

specification (local update expanded)	synch equivalent
a := **w** ∥ op(v , **w**)	-
a := **w** ; op(v , **w**)	YES

Table 5: actualIn Parameter Passing

specification (local update expanded)	synch equivalent
v := E ∥ op(**v**, w)	-
v := E ; op(**v**, w)	NO

Table 6: actualOut Parameter Passing

5 IniValueSubstitutions for Parameter Passing

Re-use of actualIn parameters we have an operation with parameters, and Expression $E \in T_2$,

$$\begin{aligned}
&\textbf{operation}\ op(\textbf{in}\ i \in T_1,\ \textbf{out}\ j \in T_2)\{\\
&\quad x \in T_1;\\
&\quad x := i;\\
&\quad j := E\\
&\}
\end{aligned}$$

We use variables v and w as actual parameter with corresponding type, and use these in a call,

$$op(v,\ w);$$

The synchronisation specification 'text' uses $\|_e$ for parallel composition of events. The sequential implementation of parallel specification is mapped left to right, and can therefore lead to the same initial_value problem that we have to deal with when mapping parallel actions to sequential implementations.
In the current approach we generate sequences in the composition by expanding the parallel composition from left to right. Therefore, in Table 5, *actualIn* parameter processing is OK, since it is the initial value of w that is used in the local assignment. The subsequent call results in a new value being assigned to w. The synchronisation $\mathbf{w} := a$ is not permitted since w can only be written once, and that occurs in the call, where w replaces the $formalOut$ parameter.

The *actualOut* parameter in Table 6 needs intial_Value substitution on entry to subroutines where they are used. We can see that the sequential implementation does not correspond to the parallel semantics since v is updated, and then used as the parameter value.

When considering formal parameters we substitute the formal the formal parameters in expressions with actual parameters in Tables 7 and 8. In Table 7 we can see that the parallel semantics are equivalent to the sequential execution for $formalOut$ parameters, but are not equivalent in Table 8 for the formal in

specification (all updates expanded)	synch equivalent
a := **w** ∥ **w** := E	-
a := **w** ; **w** := E	YES

Table 7: formalOut Parameter Passing

specification (all updates expanded)	synch equivalent
v := E ∥ **x** := v	-
v := E ; **x** := v	NO

Table 8: formalIn Parameter Passing

parameters. We apply initialValue substitution on the actualOut parameters. Tables 9 and 10 show the results this. The initial value of the parameter is stored and used in the call, the parallel semantics are retained in this way.

References

[1] E.W. Dijkstra. Guarded Commands, Non-determinacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

specification (local update expanded)	synch equivalent
v := E ∥ op(**v**, w)	-
initial_v := v ; **v** := E ; op(**initial_v**, w)	YES

Table 9: Fixed actualOut Parameter Passing

specification (all updates expanded)	synch equivalent
v := E ‖ **x** := v	-
initial_v := v; **v** := E ; **x** := **initial_v**	YES

Table 10: Fixed formalIn Parameter Passing