Verification of Event-B Event Ordering Constraints

Alexei Iliasov

Newcastle University

Plan

- motivation
- key design considerations
- language constructs summary
- an example
- demo

Motivation

Event-B does not provide means for an explicit definition of event ordering; the ordering information must be encoded in event guards

the ability to have a summary of possible event orderings in a concise and compact form help with:

- establishing functional properties of a model
- verification of use case scenarious from requirements
- code generation
- connection with other formalism (e.g., BPMN)

Motivation

the flow specification language allows a modeller to **specify** and **prove** that a given sequence of events does not contradict a given machine specification

that is, if we were to execute a machine step-by-step following the prescribed sequence of events we would not discover **divergencies** and **deadlocks** not already present in the original machine

crucially, the constraining of event ordering must be such that the overall specification is a valid **refinement** of the original model

Motivation

the ability to discharge proofs pertaining to the event ordering properties of a machine using **automated provers** is the overriding concern of the approach

we are focusing on the first-order logic provers provided in the Rodin Toolkit; they support ZF set theory and arithmetics

the limitations of the provers dictate the limits on what can be effectively expressed in the flow language

Motivation

some key requirements

- a model may not be altered (e.g., to simplify proofs)
- theorem proving is the sole verification technique (no model checkers, animators, SMT solvers, etc.)
- the solution must be compositional: proving independently that a machine satisfies two differing flows must imply that the machine satisfies the composition of the flows

Flow Language design

to see how language design decisions affect proofs let us see how the most basic ordering construct - the sequential composition operator on events - may be expressed

Flow Language design

our initial attempt is the following definition

Definition

$e_1; e_2$ means that event e_2 **immediately** follows event e_1

in other words, no other events may occur between the composed events

Flow Language design

how difficult is it to prove such a statement - $e_1; e_2$?

to exclude the occurrence of intermediate events one has to show, beside other properties, that no event other than e_2 is enabled in the after-states of e_1

this leads to n proof obligations where n is the number of machine events; it is an impractical number for any realistic model and a non-trivial flow specification let us slightly weaken the definition

Definition

$e_1; e_2$ means that event e_2 **eventually** follows event e_1

thus, although other events may interfere, it is guaranteed that the second event eventually occurs

Flow Language design

here one has to prove that an overall effect of any possible interference between the occurrences of e_1 and e_2 is such that the resultant state is a sub-state of states where e_2 is enabled

seeing all other events as relations on machine state and assuming they are already proved convergent, the effect of event interference is represented by a transitive closure of a disjunction of all interference relations

the result is a complex theorem which proof cannot be easily mechanised

Flow Language design

finally, use the following definition

Definition

$e_1; e_2$ means that event e_2 event **follow** event e_1 unless some other event happens after e_1

we only claim that it may be the case that the second event follows the first event; it may happen, however, that other event interferes and the second event is delayed or is even not reached ever in this case a condition to prove is very simple:

the after-states of e_1 must be included in the states permitted by the guard of e_2

Flow Language

e	event e
$p; q$	sequential composition
$p \| q$	parallel composition
$p \mid q$	choice
$*(p)$	terminating loop
$'start,' stop,' skip$	initialisation, termination and stuttering events

Reading flow specifications

- $first;' stop$ - after event *first* a machine may terminate
- $*(first).'stop$ - after *first* another *first* or termination
- $*(first; second).'stop$ - *second* after *first*, then *first* or termination
- $'start. * (e_1|e_2|\ldots|e_k).'stop$ - the implicit event ordering of a terminating Event-B machine

not all machine events have to be mentioned in a flow specification!

Reading flow specifications

a flow specification is nothing more than a list of theorems

for example, flow statement $f;' \, stop$ translates into

$$f;' \, stop \equiv I(v) \wedge G_f(v) \wedge S_f(v, v') \implies \bigwedge_{e \in E} \neg G_e$$

that reads as "the after-states of f (a combination of the event G_f and next-state relation S_f) are such that no other event guard is enabled"

Example

let us consider as an example a simple Event-B model of sender/receiver

we will show how to use flow specifications to check (otherwise informal) assumptions about the model

Example

$\textsc{machine}$ copy

$\quad \textsc{variables}\ \mathit{buf\_in}, \mathit{buf\_out}, \mathit{copy}$

$\quad \textsc{invariant}\ \mathit{buf\_in} \in \mathit{MSG} \wedge \mathit{buf\_out} \in \mathit{MSG} \wedge \mathit{copy} \in \mathit{MSG}$

$\quad \textsc{initialisation}\ m :\in \mathit{MSG} \| \mathit{buf\_in} := \mathit{NIL} \| \mathit{buf\_out} := \mathit{NIL}$

$\quad \textsc{events}$

$$
\begin{aligned}
\text{send} &= \textsc{any}\ m\ \textsc{where}\ m \in \mathit{MSG} \wedge \mathit{buf\_in} = \mathit{NIL}\ \textsc{then}\ \mathit{buf\_in} := m\ \textsc{end} \\
\text{recv} &= \textsc{when}\ \mathit{buf\_in} \neq \mathit{NIL}\ \textsc{then}\ \mathit{buf\_out} := \mathit{buf\_in} \| \mathit{buf\_in} := \mathit{NIL}\ \textsc{end} \\
\text{save} &= \textsc{when}\ \mathit{buf\_out} \neq \mathit{NIL}\ \textsc{then}\ \mathit{copy} := \mathit{buf\_out} \| \mathit{buf\_out} := \mathit{NIL}\ \textsc{end}
\end{aligned}
$$

$\textsc{end}$

intuitively, the following is a permissable event sequence:

$$\mathit{send}, \mathit{recv}, \mathit{save}, \mathit{send}, \ldots$$

try to check this examining the model above

Example

let us formally check the assumption that *recv* may follow *send*: *send*; *recv*

$$
\begin{array}{lcl}
\text{send} & = & \text{ANY } m \text{ WHERE} \\
 & & \quad m \in MSG \land buf\_in = NIL \\
 & & \text{THEN} \\
 & & \quad buf\_in := m \\
 & & \text{END} \\
\text{recv} & = & \text{WHEN} \\
 & & \quad buf\_in \neq NIL \\
 & & \text{THEN} \\
 & & \quad buf\_out := buf\_in \| buf\_in := NIL \\
 & & \text{END}
\end{array}
$$

theorem:

$$send; recv \equiv$$

$$I \land \overbrace{m \in MSG \land buf\_in = NIL}^{send\ \ guard} \land \overbrace{buf' = m}^{send\ \ action} \implies \overbrace{buf\_in \neq NIL}^{recv\ \ guard}$$

there is a problem: the left-hand side is too weak!

Example

$$
\begin{array}{lcl}
\text{send} & = & \text{ANY } m \text{ WHERE} \\
& & \quad m \in MSG \\
& & \text{THEN} \\
& & \quad buf\_in := m \\
& & \text{END} \\
\text{recv} & = & \text{WHEN} \\
& & \quad buf\_in \neq NIL \\
& & \text{THEN} \\
& & \quad buf\_out := buf\_in \| buf\_in := NIL \\
& & \text{END}
\end{array}
$$

indeed, the system may deadlock if m is selected to be NIL

the fix is to strengthen the guard of *send* with predicate $m \neq NIL$

Example

let us now check that that *save* always follows *recv*: *recv*; *save*

$$\begin{array}{lcl} \text{recv} & = & \text{WHEN} \\ & & \quad buf\_in \neq NIL \\ & & \text{THEN} \\ & & \quad buf\_out := buf\_in \| buf\_in := NIL \\ & & \text{END} \\ \text{save} & = & \text{WHEN} \\ & & \quad buf\_out \neq NIL \\ & & \text{THEN} \\ & & \quad copy := buf\_out \| buf\_out := NIL \\ & & \text{END} \end{array}$$

theorem:

$$recv; save \equiv$$

$$I \wedge \overbrace{buf\_in \neq NIL}^{send\ \ guard} \wedge \overbrace{buf\_out' = buf\_in \wedge buf\_in' = NIL}^{send\ \ action} \implies \overbrace{buf\_out \neq NIL}^{recv\ \ guard}$$

the theorem is OK, so we have established *send*; *recv*; *save*

Example

the next step is to demonstrate that $send; recv; save$ may be repeated for ever: check that $send$ always follows $'init$ or $save$: $'init. *(send.recv.save)$

MACHINE copy

INITIALISATION

$m :\in MSG \| buf\_in := NIL \| buf\_out := NIL$

EVENTS

$$
\begin{array}{lll}
\text{send} & = & \text{ANY } m \text{ WHERE} \\
& & \quad m \in MSG \wedge buf\_in = NIL \wedge m \neq NIL \\
& & \text{THEN} \\
& & \quad buf_i n := m \\
& & \text{END} \\
\text{save} & = & \text{WHEN} \\
& & \quad buf\_out \neq NIL \\
& & \text{THEN} \\
& & \quad copy := buf\_out \| buf\_out := NIL \\
& & \text{END}
\end{array}
$$

END

Example

the theorem is split into two cases:

- *'init* passes control to *send*:

$$Inv \wedge \overbrace{buf\_in' = NIL}^{init} \implies \overbrace{m \in MSG \wedge buf\_in = NIL}^{send\ \ guard}$$

- *save* passes control to *send*:

$$Inv \wedge \overbrace{buf\_out \neq NIL}^{save\ \ guard} \wedge \overbrace{copy' = buf\_out \wedge buf\_out' = NIL}^{save\ \ action} \implies \overbrace{m \in MSG \wedge buf\_in = NIL}^{send\ \ guard}$$

the second part cannot be discharged: the guard of *save* is too weak; the fix to strengthen it with $buf\_in = NIL$

Example: summary

even in a trivial model it is easy to make false assumptions about event ordering

model animation could help but often struggles with larger models and complex data types

bundling event flow with a model improves model readability

Demo