Flows Tool

Alexei Iliasov

Newcastle University

Contents

- Introduction
- Example 1: assumption propagation
- Example 2: scenario folding
- Example 3: deadlock-freeness and feasibility
- Generation of program counters
- Aspects
- Relative deadlock-freeness, refinement diagrams[1]
- Diagram animation[2]

[1] in version 1.1
[2] in version 1.2

Flow

The Flows plugin is an extension of the Event B Rodin platform. Its purpose is to allow a modeller to verify a range of properties related to the order of event enabling in an Event B model. The tool offers a simple graphical notation and is meant to complement the core Event B development method.

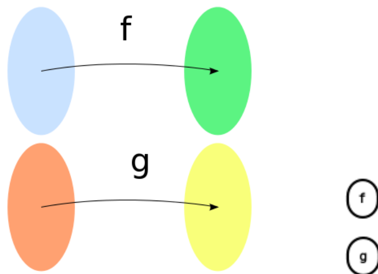
Flow

A flow diagram is a visual editor for entering certain kind of theorems. Such theorems could be added directly into a machine but this is impractical and error-prone. These set of new theorems provides an extension of the Event B proof semantics and hence gives a modeller an additional modelling tool.

An alternative viewpoint is seeing a flow diagram as a use case scenario specification. A set of such scenarious would be typically specified in system requirements and it is naturall to try to use these as verification conditions during the modelling process.

The meaning of flow POs

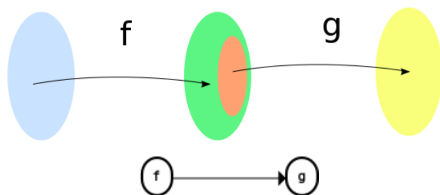
The core theorems generated by the tool characterise the relation between the after-state of one event and the guard of another.



In the figure above, two events, f and g, are characterised by their domains (guards) and ranges (after-states of actions). The righ-hand side part is the flow diagram notation.

Enabling (*FENA*)

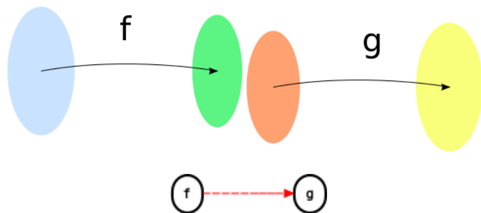
The enabling relation states that the first event enables the second event. Hence, when the first event happens it is always true that the second may happen next. Formally, the domain of the second event is fully contained in the range of the first event.



On a diagram this is denoted by a solid arrow connecting two events. Corresponding proof obligations end with *FENA* suffix

Disabling (*FDIS*)

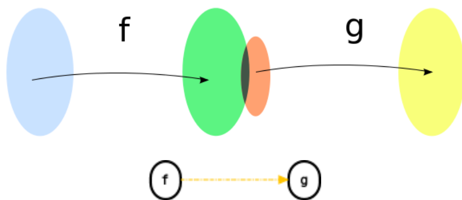
An event disables another event if the guard of the second event is guaranteed to be not enabled in the state produced by the first event.



The diagram notation is a dashed red arrow and the PO suffix is *FDIS*.

Possibly enabling (*FFIS*)

An event possibly enables another event if it potentially produces an after-state that enables the guard of the second event.



This connector is often used to check that an event is feasible in a given context. The notation is a dash-dot yellow line and the PO suffix is *FFIS*.

Combinations

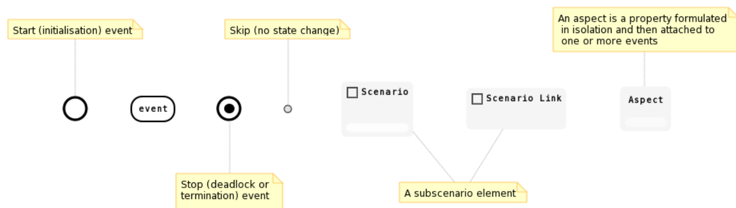
Assuming that $\mathit{FENA}, \mathit{FDIS}, \mathit{FFIS}$ denote relations as defined above, the following conditions apply.

$\neg \mathit{FENA} = \mathit{FFIS} \vee \mathit{FDIS}$	$\mathit{FENA} \wedge \mathit{FFIS} \equiv \mathit{FENA}$
$\neg \mathit{FDIS} = \mathit{FFIS}$	$\mathit{FENA} \wedge \mathit{FDIS} \equiv \bot$
$\neg \mathit{FFIS} = \mathit{FDIS}$	$\mathit{FFIS} \wedge \mathit{FDIS} \equiv \bot$

Thus, for instance, the following is a contradiction:

Diagram nodes

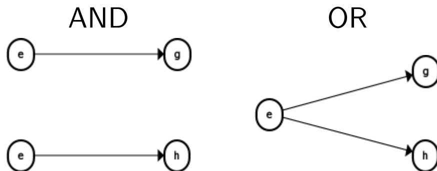
The main node class is a machine event. There is a dedicated node type for the initialisation event and nodes for implicit events *skip* and *stop*.



All further node types are used for structuring large diagrams.

Combining relations

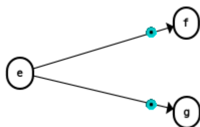
There are two different ways to combine node relations. One is to use a new instance of a source node and the other is to have several connectors originating from the same node. The former is called AND combination and results in two (or more) theorems. The latter is an OR combinations and there is just one theorem.



The general rule is that AND requires that all the specified event relations hold while for OR requires that a given relation at least for one pair of events. Different types of connectors originating from the same node do not interfere.

Relation constraints

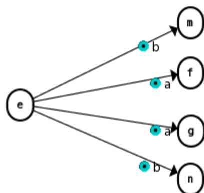
There are additional ways to constrain the OR case of the enabling relation. It is useful to be able to state that certain events are never enabled together or, on the contrary, are always enabled at the same time (in the context of the after state of a given event).



In the above, events f and g must be both enabled after the execution of event e.

Relation constraints

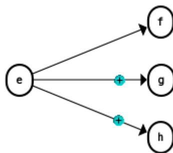
In general, one can cluster the outgoing connectors it several groups and define the constrain separately for each group.



The example diagram above states that after e events f and g must be enabled together, or, alternatively, events m and n must be enabled together.

Relation constraints

Another constraints kind is the xor-like exclusion where only one of the events is enabled. The same rules in principles apply.



Example 1

As an illustration we consider the following simple Event B model.

MACHINE calc

VARIABLES v

INVARIANT $v \in \mathbb{N}$

INITIALISATION $v := 0$

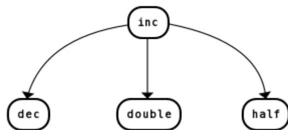
EVENTS

inc	$=$	BEGIN $v := v + 1$ END
dec	$=$	WHEN $v > 0$ THEN $v := v + 1$ END
set	$=$	ANY z WHERE $z \in \mathbb{N}$ THEN $v := z$ END
double	$=$	BEGIN $v := v * 2$ END
half	$=$	ANY u WHERE $u * u = v$ THEN $v := u$ END

END

Example 1 (1)

A simple flow diagram below states that after *inc* the system can always engage into either *dec* or *double* or *half* .

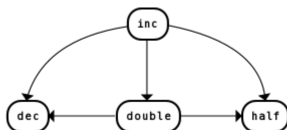


Note that there is just one theorem corresponding to the relation formed by three enabling connectors originating from event *inc*

inc/dec:double:half/FENA

Example 1 (2)

Suppose we also want to check whether *double* may be followed by *dec* or *half*.



A new theorem appears and this one cannot be discharged.

inc/dec:double:half/FENA

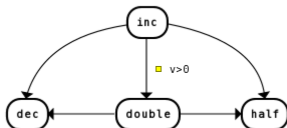
double/half:dec/FENA

Example 1 (3)

Looking closer at the *double*/*half* : *dec*/*FENA* proof obligation we would discover the need to show that either v has a natural square root (to satisfy the guard of *half*) or that v is not zero (to satisfy the guard of *dec*). There is nothing in event *double* itself to satisfy these conditions. However, from the diagram we know that, in this particular case, *double* becomes enabled after event *inc*; since *inc* assigns to v a non-zero number we should be able to satisfy the guard of *dec*.

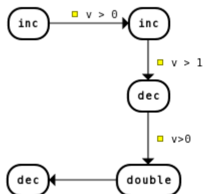
Example 1 (3)

To do the proof we have to propagate the information about the after-state of *inc* to the point where we need to prove the enabledness of *dec*. This done by placing a predicate $v > 0$ (called guard) on a connector from *inc* to *double*. For event *inc* the predicate is an assertion - the condition to proven to hold in the after-state. For *double*, it is an assumption - a condition assumed to hold whenever control is passed from *inc* to *double*.



Example 1 (3)

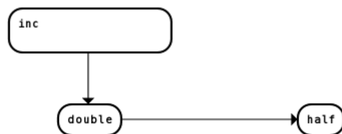
Assertion/assumption propagation is not limited to one step. One can use a complex of guards to propagate through a longer chain. Obviously, this requires more proofs as each assertion leads to an additional verification condition.



If the same property holds for a subset of a flow diagram it may be more efficient to define it using a subscenario or an aspect.

Example 1 (4)

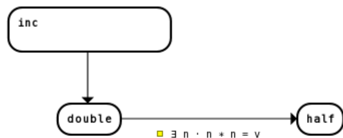
Let us focus on the more challenging branch leading from *inc* to *half* via *double* and investigate under what conditions these event sequence may take place (a reminder: *half* computes the square root of v).



It it is difficult to come up with a suitable condition straight away. A quick check shows that fitting values for v prior to the execution of *inc* look like this: $1, 7, 17, 31, \ldots$.

Example 1 (4)

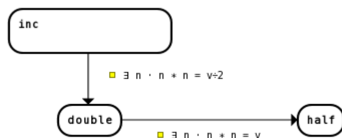
We do not need to guess what law governs these numbers. It is easily deduced from the diagram. The technique is the following. One starts by formulating an assumption that would be enough to discharge the theorem of interest. In our case, a natural assumption is that at the point when *half* becomes enabled there exists a natural number n that is a square root of v.



The addition of the assumption trivially discharges the *half* enabledness theorem.

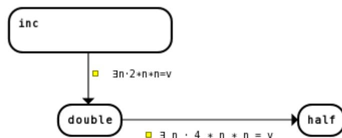
Example 1 (4)

The new assumption results in an assertion for event *double*; this cannot be proved on its own. The form of the proof suggests an additional assumption for event *double*, as shown below.



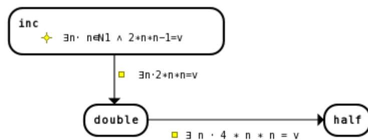
Example 1 (4)

Clearly, a natural number can be divided by two only if it is an even number. Hence, we can replace v by $v * 2$ in both guard predicates



Example 1 (4)

The new assumption gives rise to a new assertion condition, now for event *inc*. There are no incoming connection for *inc* hence we cannot push assertions any further. Instead we derive a specific instance of *inc* with a stronger enabling condition. The condition is easily computed and the overall result is shown below.



Thus, in a number of steps we have discovered set of initial states $\{2 * n^2 - 1 \| n \in \mathbb{N}1\}$ satisfying our flow diagram. The set is encoded as the constraining predicate of event *inc*.

Example 1 (5)

The constraining predicate of an event may reference event parameters. This allows one to derive an event instance with parameters constrained or even set them to concrete values.