Event Graphing and Step Inverses for CnC

Peter Elmers, Nick Vrvilo, Vivek Sarkar
The Event Graph

- Directed graph of execution flow of steps and items
- Acyclic! A cycle would be a deadlock
- Begins at an init step, ends at a finalize step
Simple Example

$context { int length; };  
[ int data: i ];

( $initialize: () ) -> [ data : 1 ], ( process: 1 );

( process: x )
   <- [ data : x ]
   -> [ data : x + 1 ],
   ( process: x + 1 ) $when(x + 1 < #length);

( $finalize: () ) <- [ data: #length ];
How it looks

length = 3:
Motivation

- Pretty pictures
- Making presentation slides
Motivation

• Debugging and understanding programs
• Confirm correctness of structural logic
Fancy example

( addToLeftEdge: row, col )
-> [ out @ cells: row, col ], ( addToLeftEdge: row+1, col );

( addToRightEdge: row, col )
-> [ out @ cells: row, col ],
   ( addToInside: row+1, col ), ( addToRightEdge: row+1, col+1 );

( addToInside: row, col )
<- [ a @ cells: row-1, col-1 ], [ b @ cells: row-1, col ]
-> [ out @ cells: row, col ],
   ( addToInside: row+1, col );
Feature overview

• Single HTML file as output

• Highlight possible errors:
  • get without put
  • put without get
  • no path to finalize
Step inverse

- A step definition represents a map from step tags to output collection tags.
  - Remark: this map is **one-to-one** if we do not output multiple items into a single collection.
  - So we can find an inverse.
What it entails

• Given a step with input tag space $X = (x_1, x_2 \ldots x_n)$,

• For each output collection with tags $T = (t_1, t_2, \ldots t_m)$ defined by mappings $t_i = f_i(x_1, x_2, \ldots, x_n)$ where $f_i: X \rightarrow \mathbb{Z}$,

• For each tag $x_k$ of the input tag space,

• Find the mapping $T \rightarrow x_k$ (i.e. solve for $x$ in terms of $T$)
Recall this one

```
$\text{context} \{ \text{int length; } \};
[ \text{int data: i} ];

( $\text{initialize: } () \) \rightarrow [ \text{data: 1} ], ( \text{process: 1} );

( \text{process: } x )
    \leftarrow [ \text{data: } x ]
    \rightarrow [ \text{data: } x + 1 ],
    ( \text{process: } x + 1 ) \text{ when}(x + 1 < \#\text{length});

( $\text{finalize: } () \) \leftarrow [ \text{data: } \#\text{length} ];
```
Simple Example

Step process:
{
'data': [{x: t1 - 1}],
'process': [{x: Piecewise(
    (t1 - 1, t1 < ctxlength), (nan, True))}]
}
What’s it for?

• Simplifies debugging

• Can blame steps for specific prescribes/puts

• Auto-blame by combining with event graph from an execution log
Stenciling

$context \{ \text{int lastTile; int numIters; } \};$

// tile at index $i$, iteration (timestep) $t$
[ float *tile: $i$, $t$ ];

( $initialize: ()$ )

$\rightarrow$ [ tile: $\text{rangeTo(#lastTile), 0}$ ],
( stencil: $\text{rangeTo(#lastTile), 0}$ );

( stencil: $i$, $t$ )

$\leftarrow$ [ prevT @ tile: $i$, $t$ ],
[ prevL @ tile: $i -1$, $t$ ] $\text{when}(i > 0)$,
[ prevR @ tile: $i +1$, $t$ ] $\text{when}(i < \text{lastTile})$
$\rightarrow$ [ nextT @ tile: $i$, $t +1$ ],
( stencil: $i$, $t +1$ ) $\text{when}(t +1 < \text{numIters})$;

( $finalize: ()$ )

$\leftarrow$ [ tile: $\text{rangeTo(#lastTile), numIters}$ ];
How it looks
( $initialize: () )

-> [ tile: $rangeTo(#lastTile), 0 ],
   ( stencil: $rangeTo(#lastTile), 0 );

( stencil: i, t )

← [ prevT @ tile: i, t ],
   [ prevL @ tile: i-1, t ] $when(i >= 0),
   [ prevR @ tile: i+1, t ] $when(i < #lastTile)

→ [ nextT @ tile: i, t+1 ],
   ( stencil: i, t+1 ) $when(t+1 < #numIters);

( $finalize: () )

← [ tile: $rangeTo(#lastTile), #numIters ];
Look at the graph
Performing blame on these nodes: ['tile@-1, 0']

Blaming tile@-1, 0: {'stencil': {'i': -1, 't': -1}}
In the future...

- Demand-driven execution:
  - Specify the output, and solve backwards to the initial step
  - All steps visited should be prescribed
What that means
Wrap up

• Event graphs and step inverses
• Tool assisted debugging
• A peak at the future