Spatio-temporal modelling and simulation with the FCPP Aggregate Programming framework

II - FCPP

Volker Stolz,[†] Ferruccio Damiani,[*] Giorgio Audrito[*]

[*]University of Turin, Italy
[†]Western Norway University of Applied Sciences, Bergen

June 20, 2022

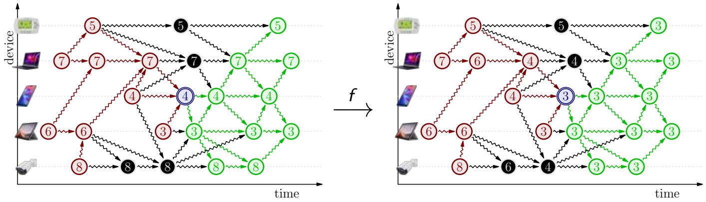
Outline

- Implementing the model
  - An abstract model
  - A simple concrete model
- The aggregate language
- Sample aggregate programs
- The FCPP platform

An abstract model

formal model of distributed computation

- event structures: E events, $\rightsquigarrow$ message DAG and $<$ causality partial order
- space-time values $\Phi : E \to X$ are labelling (functions) of event structures
- composable space-time functions $f(\Phi_{\text{in}}) = \Phi_{\text{out}}$ are functions on s-t values

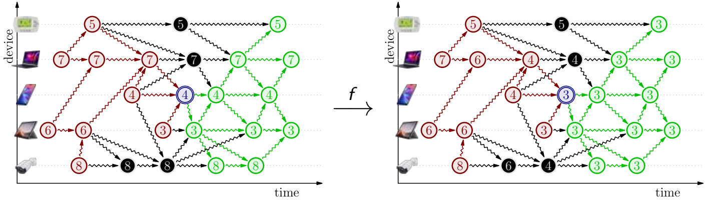


- programs built by composing functions with nice global interpretation
- . . . we need a way to express basic functions via local interactions

An abstract model

formal model of distributed computation

- the abstract model doesn't care how the functions may be implemented
- ...even C programs sending custom network packets as they please
- ...but composition in C doesn't match with model composition ☹



better try to use something more high-level!

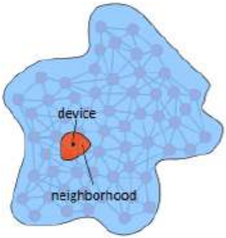
A simple concrete model

simplifying assumptions...

- the same program is executed in every event
- ...can still execute different code through branching
- messages are sent through broadcast (can extend to pointwise messages)

Round:

1. gather data received, stored and sensed
2. compute the program
3. broadcast the result to neighbours
4. perform actuation as computed
5. receive messages while sleeping

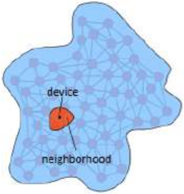


Outline

- Implementing the model
- The aggregate language
  - Language features
  - Coordination constructs: `old`
  - Coordination constructs: `nbr`
  - Coordination constructs: `if`
- Sample aggregate programs
- The FCPP platform

Language features

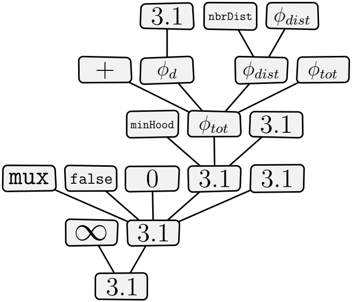
- implicit messages via constructs with automatic send/receive matching
- . . . using received data to generate the message to send
- language composition matching with model composition
- **[universality]** shows that we can still express all s-t functions



. . . but how?

- using evaluation trees to guide matching
- modelling messages with neighbouring field values

Language features



evaluation trees (value-tree for short)

- in functional programming, a program is an expression that is evaluated
- the resulting value is generated by evaluating sub-expressions
- overall it produces an ordered tree of values

implicit messages matching

- messages are identified with the nodes in the tree originating them
- . . . basically with their stack trace
- different branches of `if` are considered different
- different call, different traces, no interference
- works well with function composition and branching!

Language features

messages as neighbouring field values

- $\mathtt{v}$ are usual values (Boolean, integer, tuple...)
- $\phi = \overline{\delta} \mapsto \overline{\mathtt{v}}$[a] is a map from device identifiers of neighbours $\overline{\delta}$ to values $\overline{\mathtt{v}}$
- we say that ϕ has type $\mathtt{field}{<}T{>}$ if all values $\overline{\mathtt{v}}$ have type T
- regular values can be interpreted as constant neighbouring field values
- ϕ represents incoming or outgoing messages
- functions are overloaded to act pointwise on neighbouring field values: $\phi_1 + \phi_2 = \phi_3$ s.t. $\phi_3(\delta) = \phi_2(\delta) + \phi_1(\delta)$ for all δ
- binary functions can be used to aggregate neighbouring field values: $\mathtt{fold\_hood}(\min, \phi) = \mathtt{min\_hood}(\phi) = \min\{\phi(\delta) : \ \delta \text{ neighbour}\}$

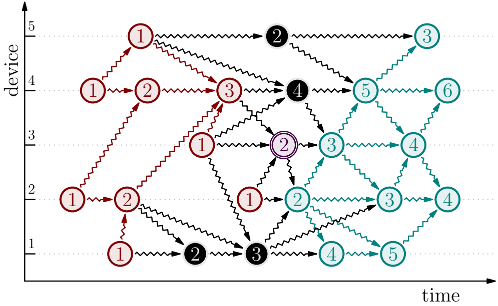
[a] we use $\overline{x}$ to denote the sequence $x_1, \ldots, x_n$.

Coordination constructs: old(CALL, e)

- stores the current value of e
- . . . and returns the value passed in the previous round

$$\texttt{old}(\texttt{CALL}, \texttt{node.position}())$$

retrieves the position that the node had in the previous round.



e $\longrightarrow$ 2, store 2

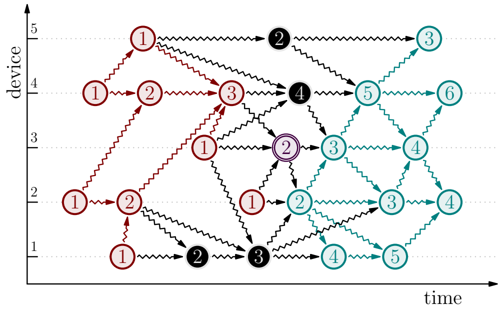
$\texttt{old}(\texttt{CALL}, \texttt{e}) \longrightarrow 1$

Coordination constructs: $\mathtt{old}(\mathtt{CALL}, \mathtt{e}_i, \mathtt{e}_f)$

- represents state evolution between rounds of computation
- starts from an initial value $\mathtt{e}_i$
- repeatedly alters the state using an update function $\mathtt{e}_f$

$$\boxed{\mathtt{e}_c := \mathtt{old}(\mathtt{CALL}, 0, [\&](\mathtt{int}\ \mathtt{x})\{\mathtt{return}\ \mathtt{x} + 1; \})}$$

counts how many computational rounds have elapsed since the beginning.



$\mathtt{e}_c \longrightarrow$

$[\&](\mathtt{int}\ \mathtt{x})\{\mathtt{return}\ \mathtt{x} + 1; \}(1)$

$\longrightarrow 1 + 1$

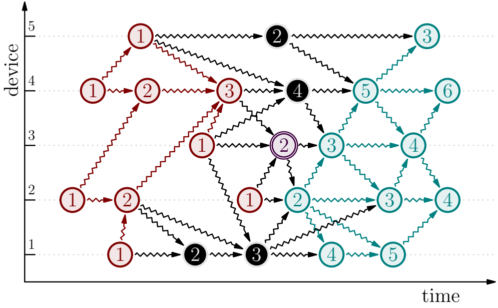
$\longrightarrow$ 2, store 2

Coordination constructs: $\mathtt{nbr}(\mathtt{CALL}, \mathtt{e})$

- represents interaction between neighbour devices
- sends result of e to neighbours (duality outgoing - incoming)
- collects neighbour's values for the same e into a neighbouring field

$$\mathtt{sum\_hood}(\mathtt{CALL}, \mathtt{nbr}(\mathtt{CALL}, 1));\ \mathtt{nbr}(\mathtt{CALL}, \mathtt{e}_c)$$

counts the number of neighbours; *neighbouring field of counters.*



$\mathtt{e}_c \longrightarrow 2$, broadcast 2

$\mathtt{nbr}(\mathtt{CALL}, \mathtt{e}_c) \longrightarrow \phi$ where

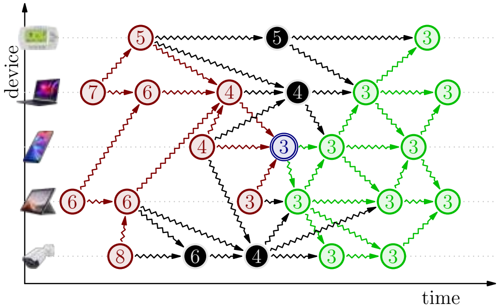
$\phi = \delta_2 \mapsto 1, \delta_3 \mapsto 2, \delta_4 \mapsto 3$

Coordination constructs: $\mathtt{nbr}(\mathtt{CALL}, \mathsf{e}_i, \mathsf{e}_f)$

- represents shared state evolution
- works as `old` (e_i initial value, e_f update expression)
- e_f is applied to the neighbouring field of values shared by neighbours
- result is shared with neighbours in return

$$\mathsf{e}_g := \mathtt{nbr}(\mathtt{CALL}, v, [\&](\mathtt{field}{<}\mathtt{int}{>}\ \mathtt{x})\{\mathtt{return}\ \mathtt{min\_hood}(\mathtt{CALL}, \mathtt{x}, v)\})$$

gossips the minimum v in a network.



$\mathsf{e}_g[v := 4] \longrightarrow$

$[\&](\mathtt{field}{<}\mathtt{int}{>}\ \mathtt{x})\{\mathtt{return}$
$\mathtt{min\_hood}(\mathtt{CALL}, \mathtt{x}, 4)\}(\phi)$

$\longrightarrow \mathtt{min\_hood}(\mathtt{CALL}, \phi, 4)$

$\longrightarrow \mathtt{min}(3, 4) \longrightarrow 3$

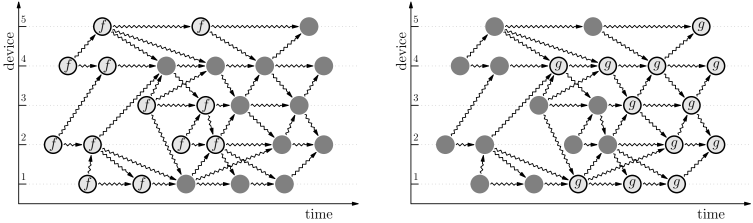
$\phi = \delta_2 \mapsto 3, \delta_3 \mapsto 4, \delta_4 \mapsto 4$

Coordination constructs: $\texttt{if(e)}\{\texttt{e}_\top\}\,\texttt{else}\,\{\texttt{e}_\bot\}$

- $\texttt{nbr(CALL, e)}$ collects values among neighbours which calculated the same $\texttt{e}$
- within a branch, if a neighbour chose another branch it is not considered
- the result is a restriction of the domain of the neighbouring field
- ...is a feature enabling computation in sub-networks

$$\texttt{if}(\textit{active})\{\texttt{program}();\}$$

executes a program only on the sub-network of active devices.



Outline

Basic Examples

A program that computes whether temperature is high

```
node.storage(temperature{}) > 20
```

A program that shows whether temperature is high

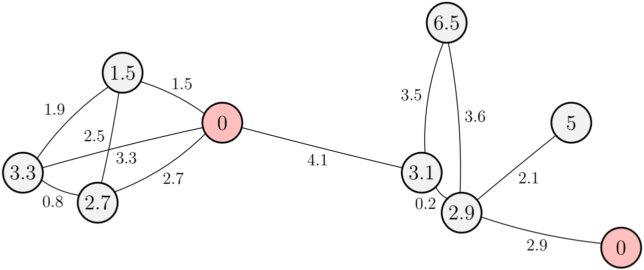
```
if (node.storage(temperature{}) > 20) {
        node.storage(led{}) = color(GREEN);
} else {
        node.storage(led{}) = color(ORANGE);
}
```

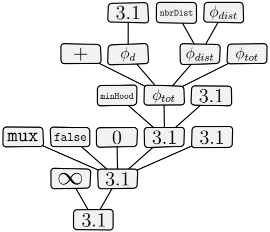
Another example: Adaptive Bellman-Ford

implementation with old and nbr

```
FUN real_t distanceTo(ARGS, bool source) { CODE
  return nbr(CALL, INF, [&](field<int> d){
    real_t nd = min_hood(CALL, d + node.nbr_dist());
    return source ? 0 : nd;
  });
}
```

`node.nbr_dist` returns a neighbouring field of distances from neighbours

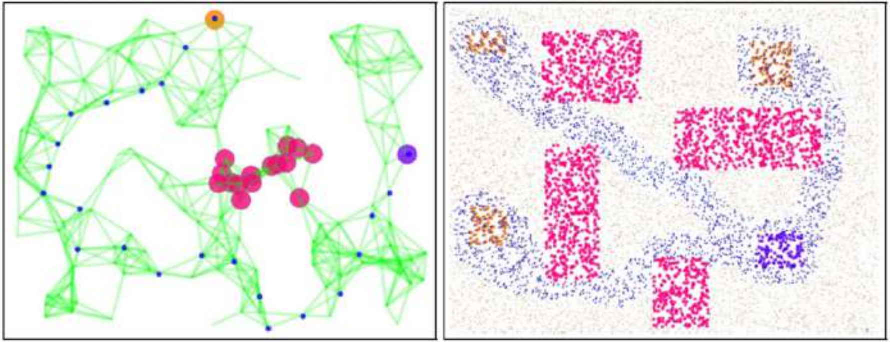




One final example: channel with obstacles (simulation)

Channels (blue) route between source (orange) and destination (purple) around obstacles (pink), deployed in

- a low-density network with topology (green) in evidence (left), and
- in a high-density environment of 10,000 nodes (right)



A function that broadcasts a value from a source

```
FUN real_t broadcast(ARGS, real_t dist, real_t value) { CODE
  return get<1>(nbr(CALL, make_tuple(dist, value), [&](auto nx){
    return min_hood(CALL, nx, make_tuple(dist,value));
  }));
}
```

A function that makes a distance available everywhere

```
FUN real_t distance(ARGS, bool source, bool destination) { CODE
  return broadcast(distanceTo(CALL, source), distanceTo(CALL, destination));
}
```

A function that computes a channel

```
FUN bool channel(ARGS, bool source, bool destination, real_t width) { CODE
  return distanceTo(CALL, source) + distanceTo(CALL, destination)
    < width + distance(CALL, source, destination)
}
```

A function for a channel avoiding obstacles

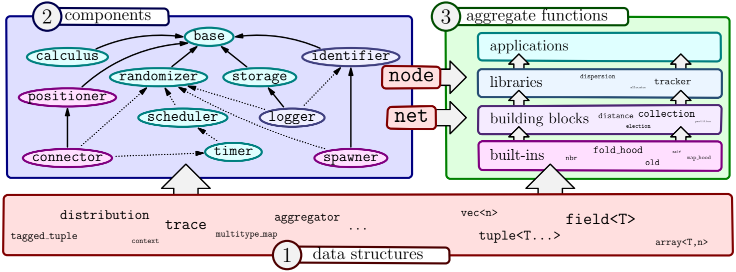
```
FUN bool channelAvoidingObstacles(ARGS, bool o, bool s, bool d, real_t w) { CODE
   if (o) { return false; } else { return channel(CALL,s,d,w); }
}
```

Outline

- Implementing the model
- The aggregate language
- Sample aggregate programs
- The FCPP platform
  - main features
  - application scenarios
  - simulation features
  - language features

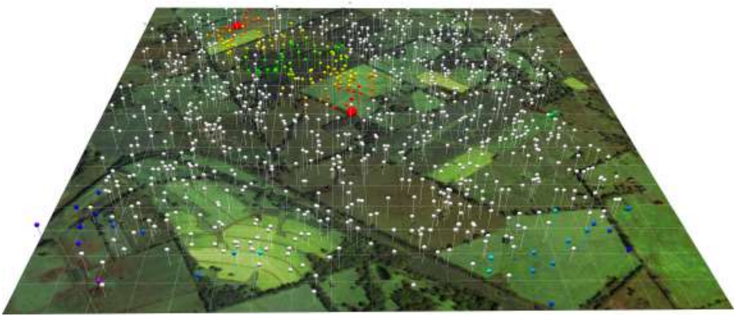
FCPP main features

- C++ library used to develop distributed programs using it
  - manipulates C/C++ values
  - can use external C/C++ code
  - portable to any architecture with C++ compiler
- performance optimized at compile-time
- extensible component-based architecture



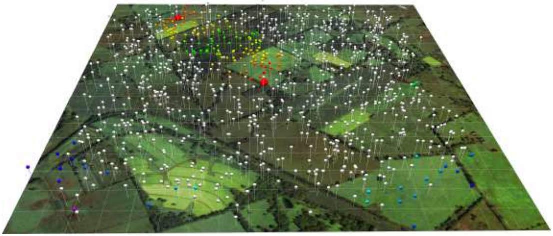
FCPP application scenarios

- dynamic HPC computations
- microcontroller (WSN) distributed networks (Miosix, Contiki)
- batches of simulations of WSN
- 3D interactive graphical simulations of WSN



FCPP simulation features

- 2D and 3D physics with acceleration and friction
- multiple wireless connection models
- random or regular geometric deployments
- support for map navigation



FCPP language features

- basic programming syntax inherited from C/C++
- special `node` object to access built-in functions and simulator features
- aggregate functions (`field<T>`, `old`, `nbr`...)
- `FUN`, `ARGS`, `CODE`, `CALL` macros
- library of coordination functions in the `coordination` namespace
- documentation available online