Introduction to Behavioral Programming

Based on research in David Harel's group

Assaf Marron

Weizmann Institute of Science

Background: Software Engineering for Reactive Systems

» Decades of advances in languages, tools and methodologies

» Growing demand, criticality of software systems

» Yet, software development is still hard, expensive, risky:

> Requirements gathering/understanding/encoding

> System design/structure

> Testing and verification

> Maintenance

> . . .

“The ‘software crisis’ stifles innovation”

(Chen Kaesar)

Goal: "Liberating Programming" (Harel 2008)

Develop complex software systems

from simple specifications of desired behavior

- ✓ You shall do …
- ✓ You shall not do …
- ✓ . . .

that are interwoven automatically

The Goal: "Liberating Programming"

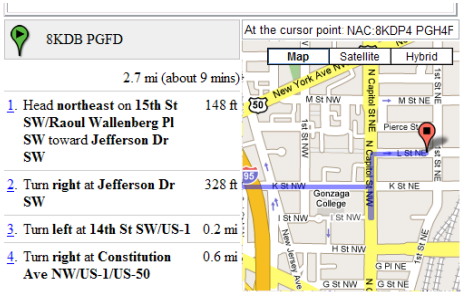
In reactive systems where complexity comes from the need to interweave many simultaneous behaviors and exceptions

we want to
enable development with

components that are aligned with how people describe behavior

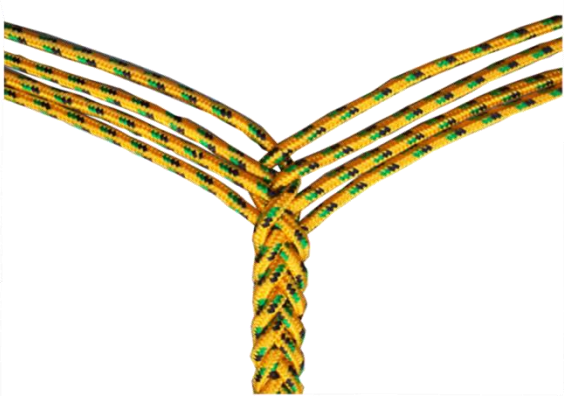
Humans interweave behaviors naturally all the time

Route



- Go for 90 km on road A1
- Turn left to road A4
- Go for 70 km on road A4
- . . .

+



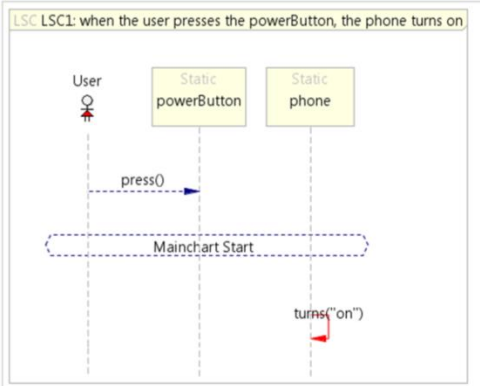
Schedule

- . . .
- Drive for 4 hours
- Stop for lunch
- Drive for 3 hours
- Look for a hotel
- Stop for the night

= Trip

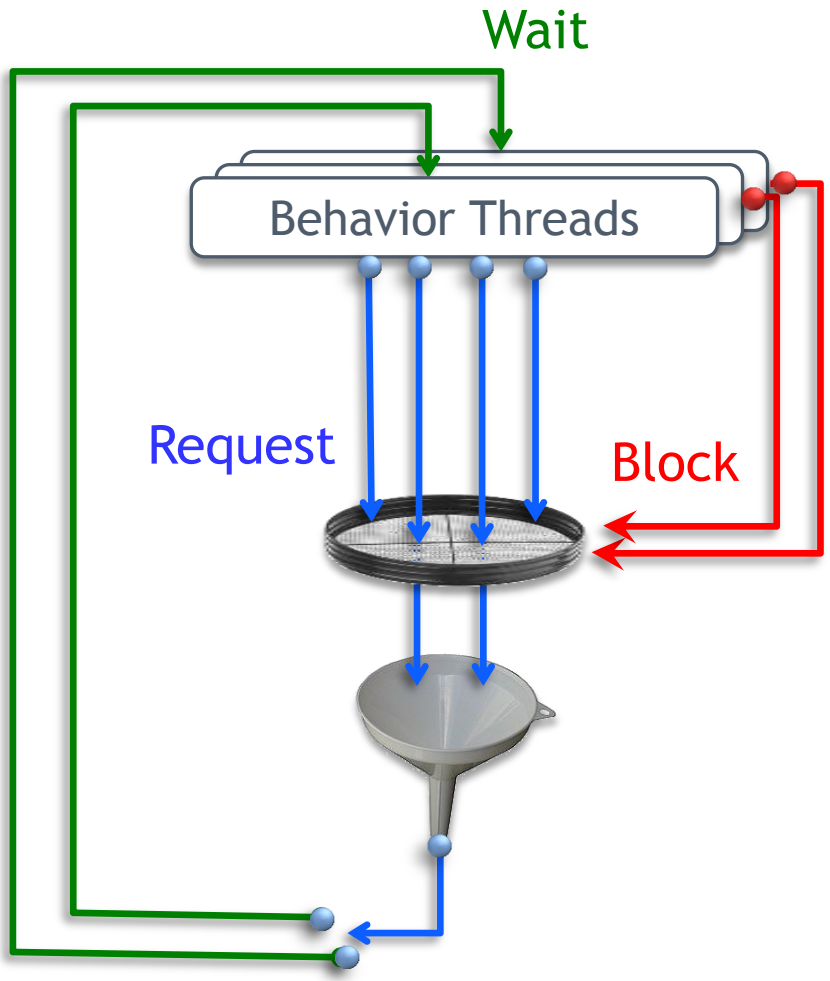
Behavioral Programming (BP)

- Program modules are scenarios of system behavior (behavior threads)
- All scenarios run simultaneously
- All are consulted at decision points during execution
- **Supported in various languages:**
  - Started with the visual language of *live sequence charts* - LSC:
    [Damm & Harel 2001],
    [Harel & Marelly 2003],
    [Maoz & Harel, 2006],
    [Harel & Maoz & Szekely & Barkan 2010]
  - Java: [Harel & Marron & Weiss, 2010]
  - C, C++, JavaScript, Blockly, Erlang…

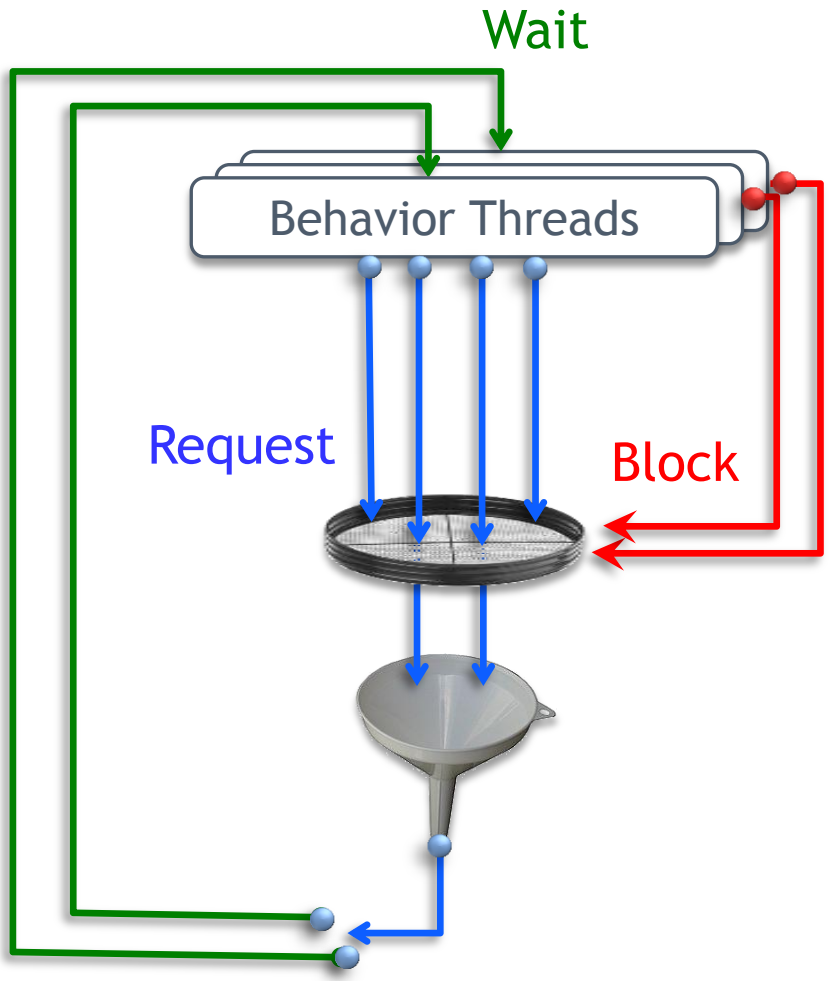



```
bSync(Event1, Event2, Event3);
```

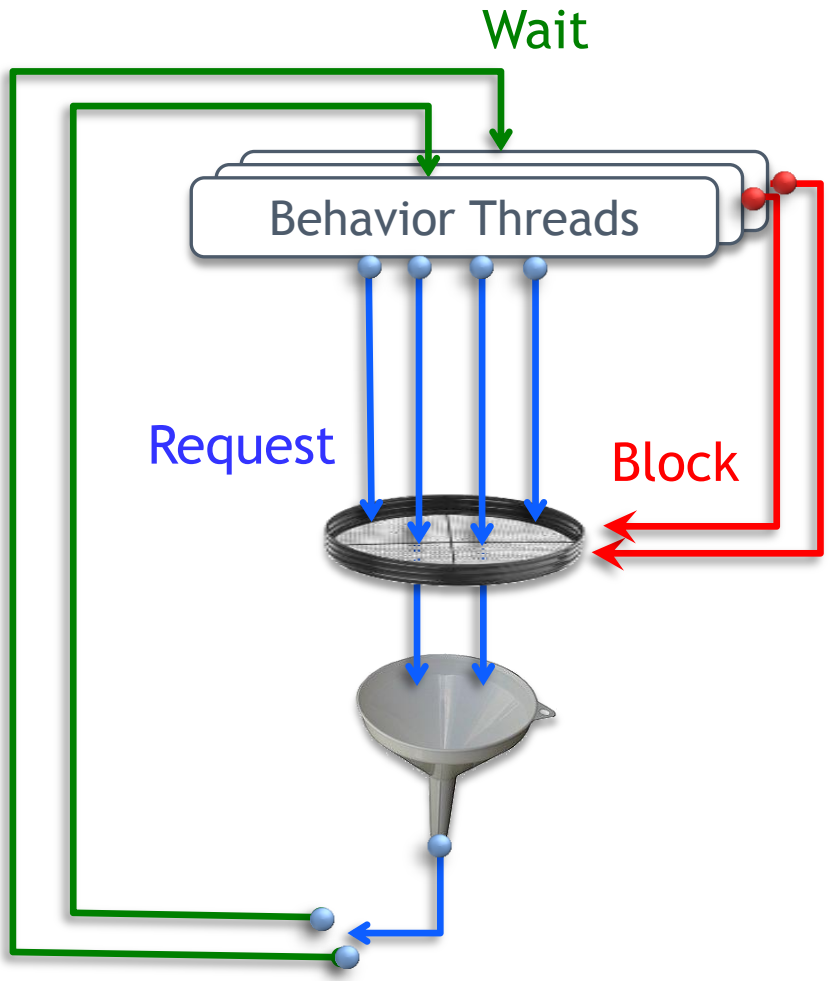
The BP Execution Cycle



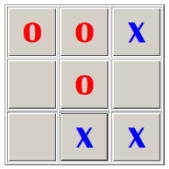
The BP Execution Cycle



The BP Execution Cycle



A new programming idiom: forbidden behavior (event blocking)



Enforcing player turns in a game

```
do forever {
   Block Player1 moves until Player2 makes a move;
   Block Player2 moves until Player1 makes a move;
}
```

Controlling a quadrotor

A behavior for each desired parameter (angle/altitude/speed):

```
do forever {
   Request actions contributing towards desired value
   while blocking opposite actions
}
```

Interweaving independent threads

```
AddHotFiveTimes() {
   for i=1 to 5 {
      bSync(request=addHot, wait-for=none, block=none);
   }
}
```

```
AddColdFiveTimes() {
   for i=1 to 5 {
      bSync(request=addCold, wait-for=none, block=none);
   }
}
```

```
Interleave() {
   forever {
      bSync(request=none, wait-for=addHot, block=addCold);
      bSync(request=none, wait-for=addCold, block=addHot);
   }
}
```

addHot
addCold
addHot
addCold
addHot
addCold
addHot
addCold
addHot
addCold

Examples (see movies on web site)

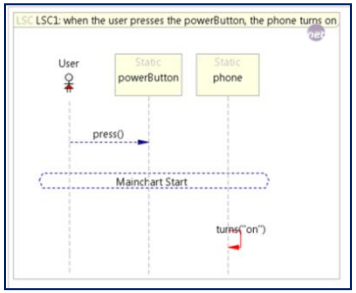
- Flying a helicopter:
  - Mixing missions
  - Correcting location
  - Stabilizing

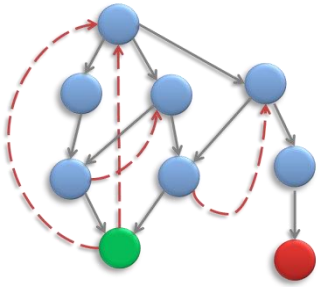
- Playing games
  - Separate scenarios for rules, strategies

- Simulating flock of birds
  Emergent behavior from multiple simple behaviors

Theory and Tools for Reactivity

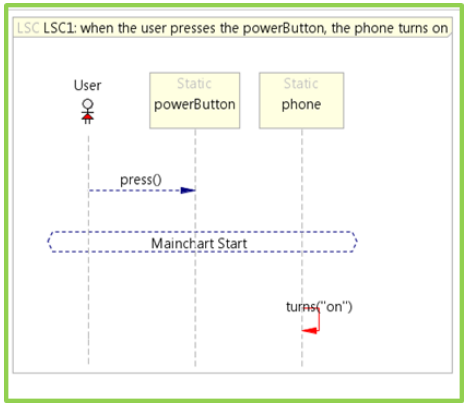
- Languages and programming
- Execution
- Verification
- Natural language input
- Program comprehension
- Adaptivity and learning
- 2D and 3D visualization
- Applications/demonstrations
- CS Education
- More . . .





```
waitFor(Event1);
waitFor(Event2);
blocking (Event3)
        request(Event4)
```

Why use BP?



1. Naturalness in development

- **Structure**: Requirements, bug reports, etc., can be mapped to program modules

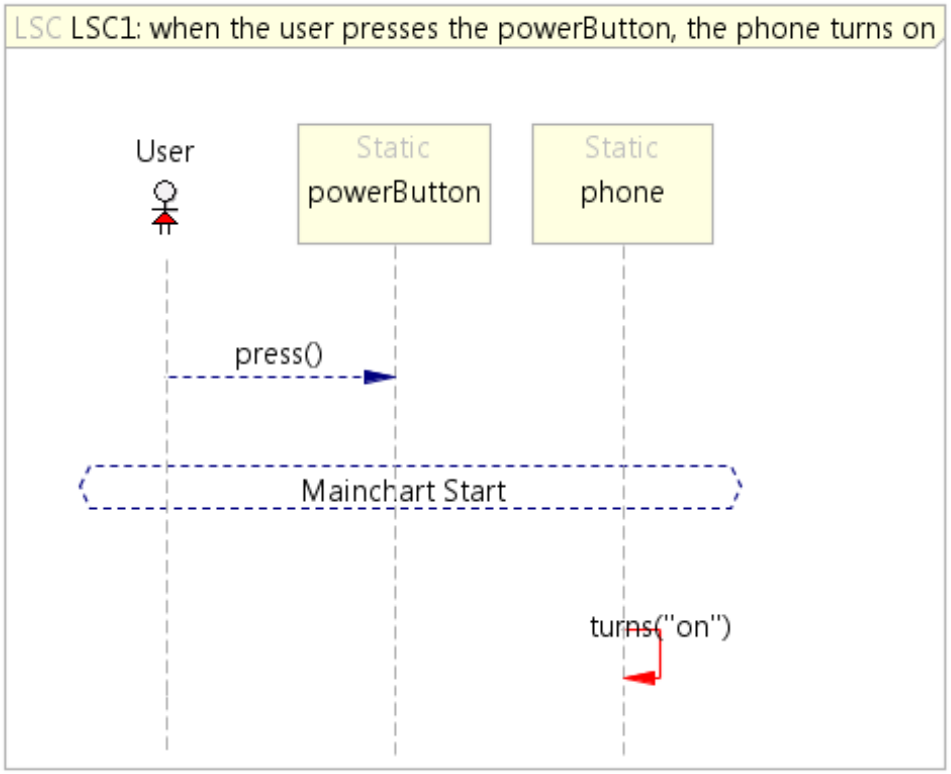
  [Damm & Harel, 2001], [Harel & Marelly, 2003]

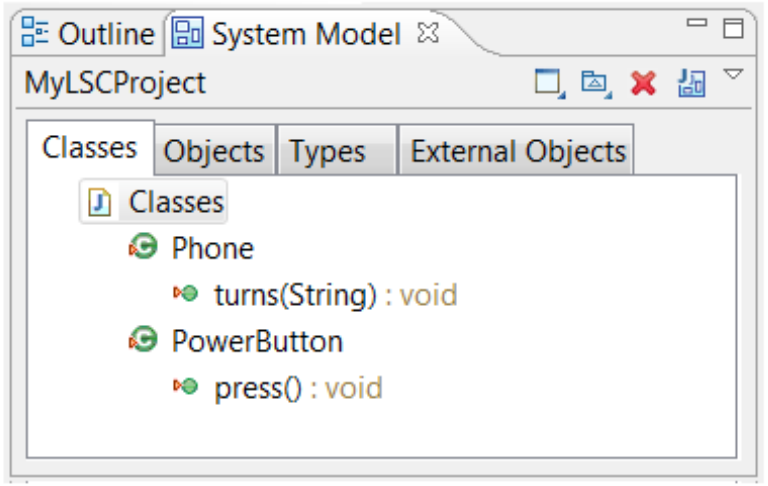
- **Intuitive programming:** visual, natural language, scenarios

  [Harel & Kugler & Marelly, 2002], [Gordon & Harel, 2009]
  [Gordon & Marron & Meerbaum-Salant, 2012]

- Resulting in fully executable specifications that can serve in simulators and in final systems

Visual and NL programming with the PlayGo tool





when the user presses the powerButton, the phone turns on

[Harel & Maoz & Szekely & Barkan, 2010], [Gordon & Harel, 2009]

Why use BP?

2. Incremental system evolution

- **New requirements, enhancements and bug repair "patches", affecting overall system behavior, can be added with little or no change to existing modules**

[Damm & Harel, 2001],
[Harel & Marelly, 2003],
[Harel & Lampert & Marron & Weiss, 2011]
[Harel & Katz & Marron & Weiss, 2012]

Why use BP?

3. Amenability to automated smart execution, verification, and synthesis

- **Tool support:** Formal executable semantics enables direct use of the tools, e.g., to find conflicts in original requirements

  [Harel & Kugler & Marelly & Pnueli, 2002],
  [Harel & Lampert & Marron & Weiss, 2011],
  [Harel & Segall, 2011],
  [Maoz & Saar, 2013]

- **Compositionality:** Application-agnostic composition can enable efficient inference of system properties from module properties

  [Harel & Kantor & Katz & Marron & Mizrahi &Weiss, 2013]

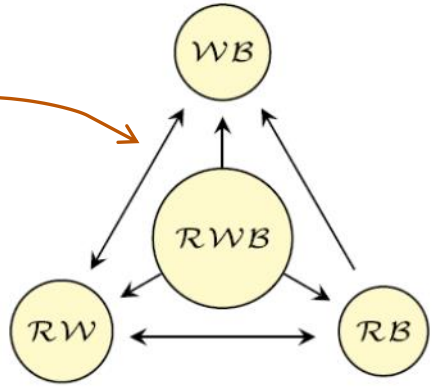
Why use BP?

4. Succinctness – possibility of small modules

- **Proved that with BP some programs can be built from parallel modules that are exponentially smaller, in terms of number of states in a transition system, than with other programming idioms**
- **BP offers some of the succinctness advantages of cooperating automata (statecharts) using interfaces that preserve more encapsulation.**

[Harel & Lampert & Katz & Marron & Weiss, 2013]

Each arrow means existence of languages $\langle L_n \rangle_{n=1}^{\infty}$ that can be expressed with some idioms exponentially more succinctly than with others



BP challenges
and
directions for answers

- BP may not be needed or desired
  - when you have a simple reactive decision
  - when incrementality is not needed

- Conflicts → Verification helps find conflicting requirements
- Comprehension → Debugging and visualization tools
- Performance → Hardware support ; automated synthesis
- Scalability → Hierarchy, continuous processes/CPS

Our research is aimed at showing applicability, removing obstacles

Behavioral Programming Summary

- A language independent direction for liberating programming
- Implemented in LSC, Java, C++, JavaScript, Blockly, Erlang, and more
- Application modules are the required and forbidden scenarios
- All scenarios run in parallel and all are constantly consulted
- Benefits:
  - Naturalness
  - Incrementality
  - Amenability to automated (compositional) analysis
  - Succinctness
  - More…
- Tools for programming, verification, comprehension, integration
- Openly available
- Research continues

The Team

The group: http://www.wisdom.weizmann.ac.il/~harel/research.html
and many more collaborators at Weizmann, Ben Gurion Univ. and worldwide.

Additional Notes

Automatic Patching

- Safety properties: nothing bad happens
  - For instance: the program never loses the game
- There are properties over the program's events
  - For instance: after event e_1, event e_2 can't be triggered
- Violating runs are found using a model checker
  - The runs are lists of triggered events: $e_1, e_2, \ldots, e_n$
- A new thread is added to the program:
  - Wait for $e_1, e_2, \ldots, e_{n-1}$, then block e_n and terminate
- The Event Selection Mechanism will choose a different route

Depth Limitation

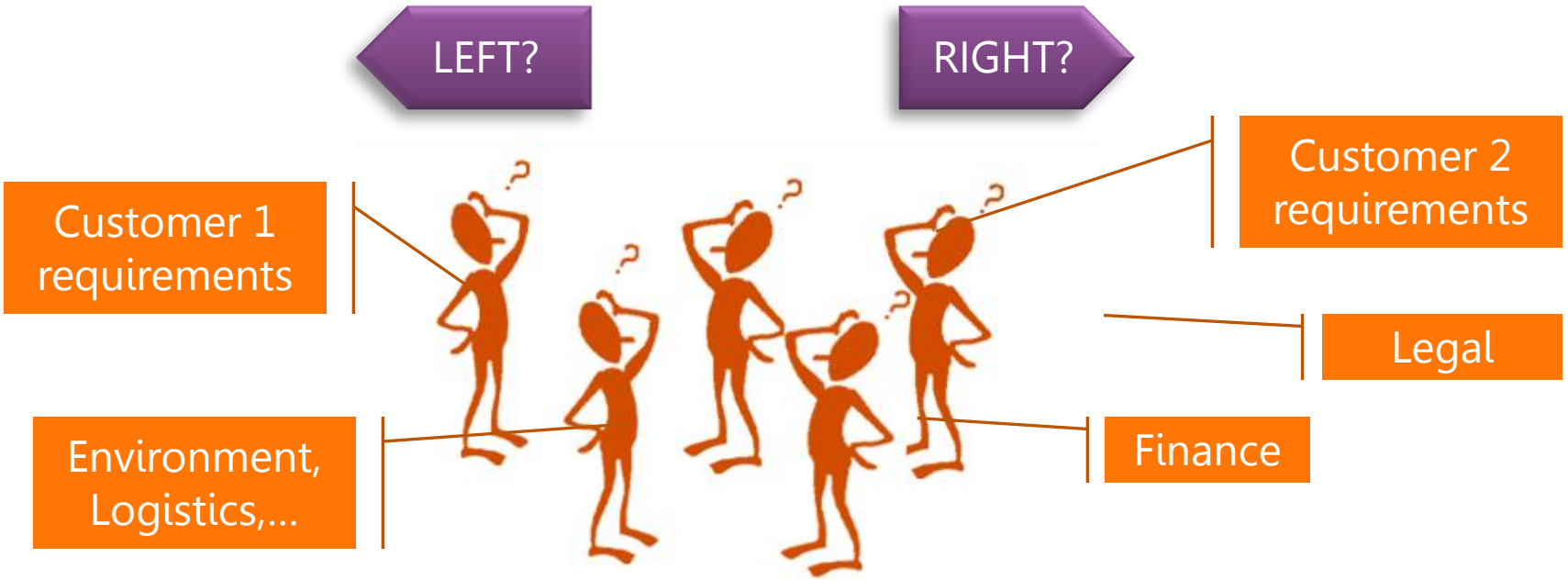
- Instead of running a full model checker, collect user bug reports and apply limited depth model checking
  - The user submits a log containing a violating run

  - We model check a bounded neighborhood of that run
  - For each violation found, we proceed as before
- Fixes the reported bug and other bugs close by
  - Far away bugs go undetected
- Not perfect, but more practical for complex programs

BP Execution

At every decision point in execution different components represent different facets of the next decision

A general interweaving mechanism collects and evaluates the information and then makes a decision

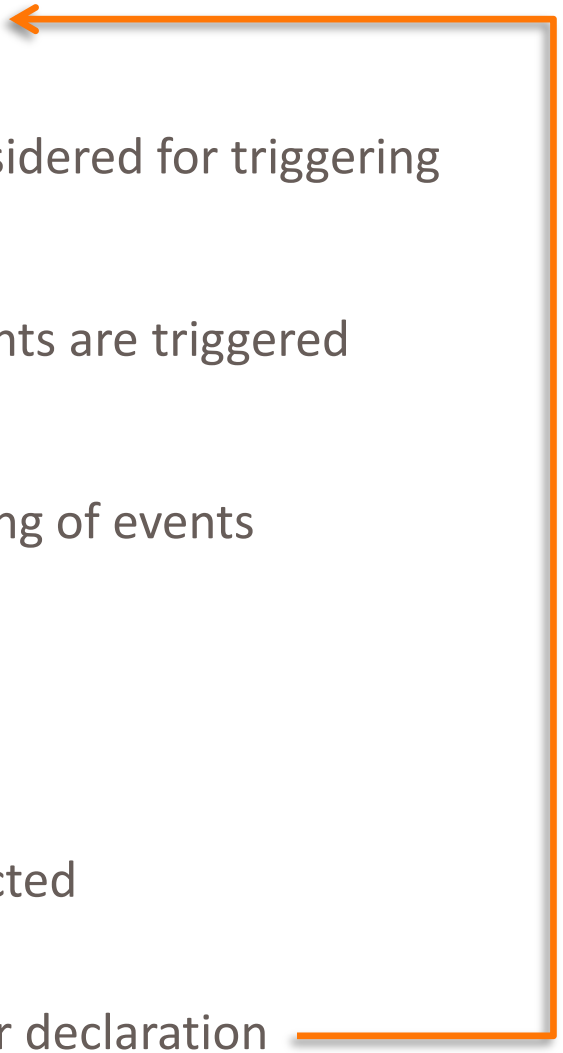


Execution Cycle in Behavioral Programs

1. All behavior threads (b-threads) post declarations:
   - Request events: propose events to be considered for triggering
   - Wait for events: ask to be notified when events are triggered
   - Block events: temporarily forbid the triggering of events
2. When all declarations are collected:

   An event that is **requested** and not **blocked** is selected

   All b-threads **waiting** for this event can update their declaration



Stabilizing a Helicopter

Current

Program incrementally - each change in angle or altitude

Do not speed up rotor X

OK to slow down rotor X

OK to speed up Rotor Y

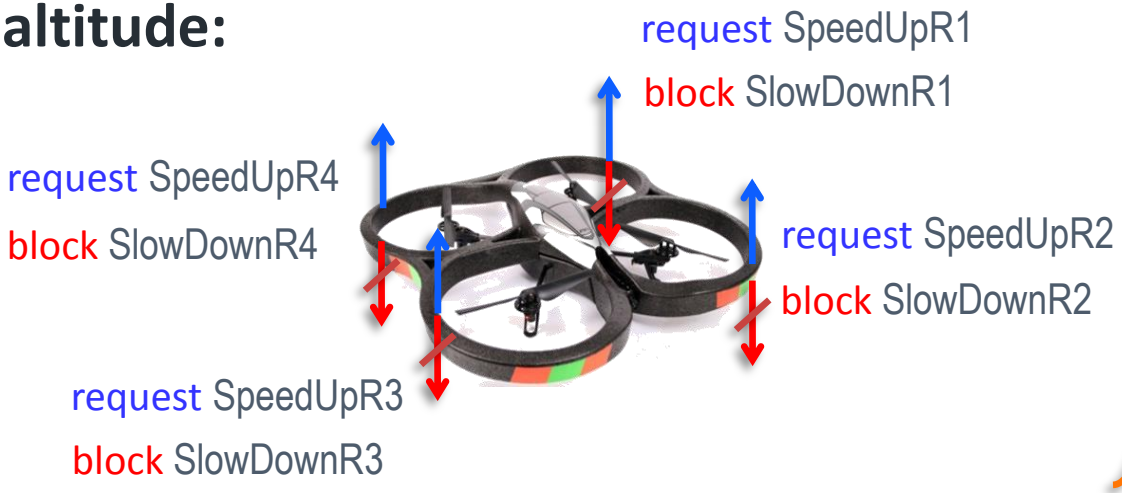
Do not slow down Rotor Y

Desired

Example: Flying a quadrotor helicopter

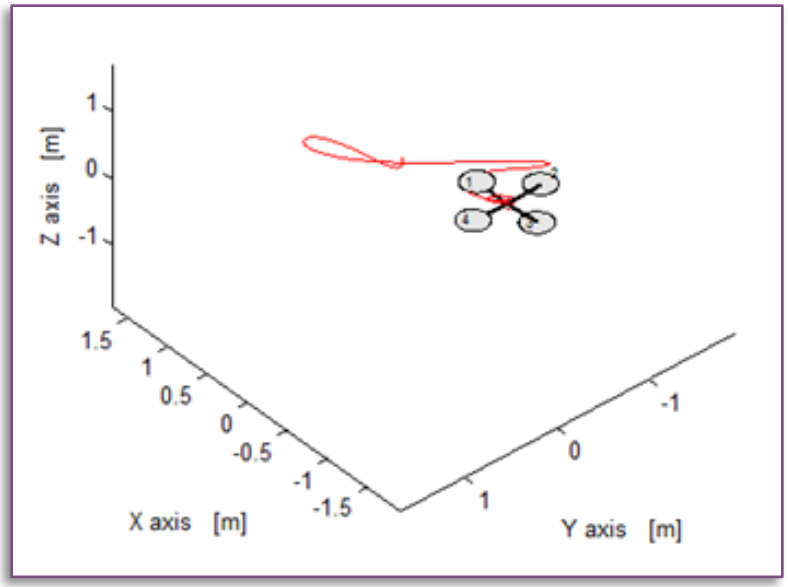
To correct the angle:

To increase altitude:



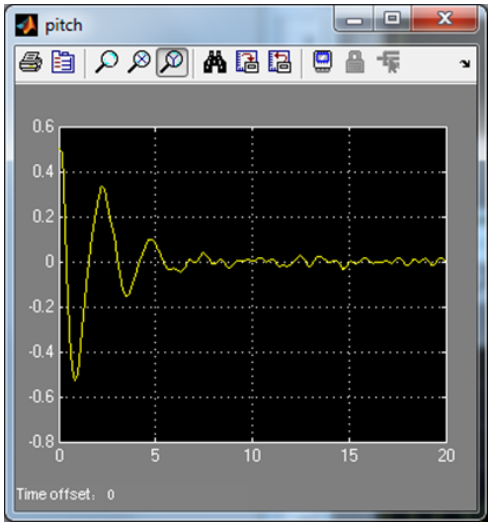
Selected event: **SpeedUpR2**

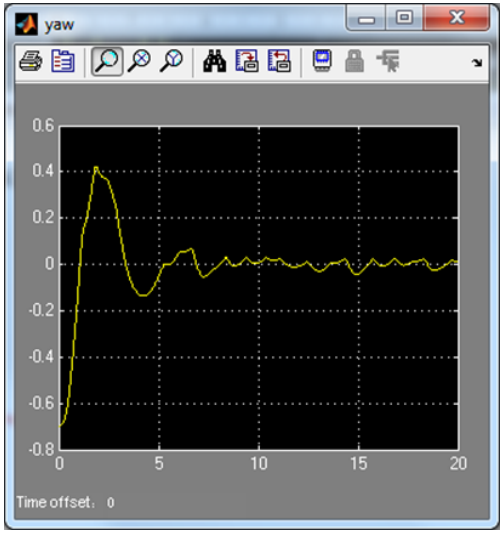
Stabilizing a quadrotor - behaviorally

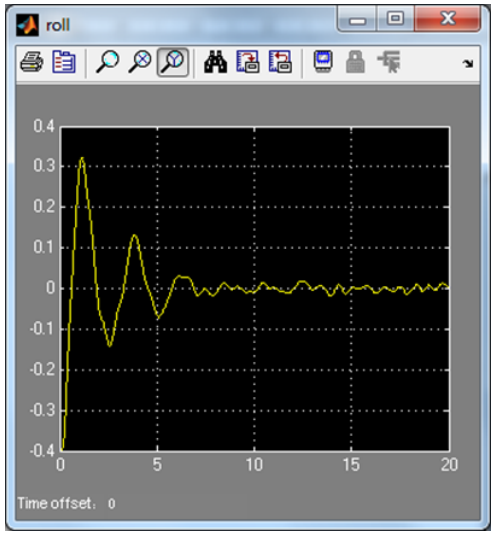


Results of applying the quadrotor Simulink model of Bouabdalla et al where a linear transformation box is replaced with behavior threads.

Independent b-threads integrated at run time stabilize the UAV in a few seconds







A behavior thread

A b-thread:

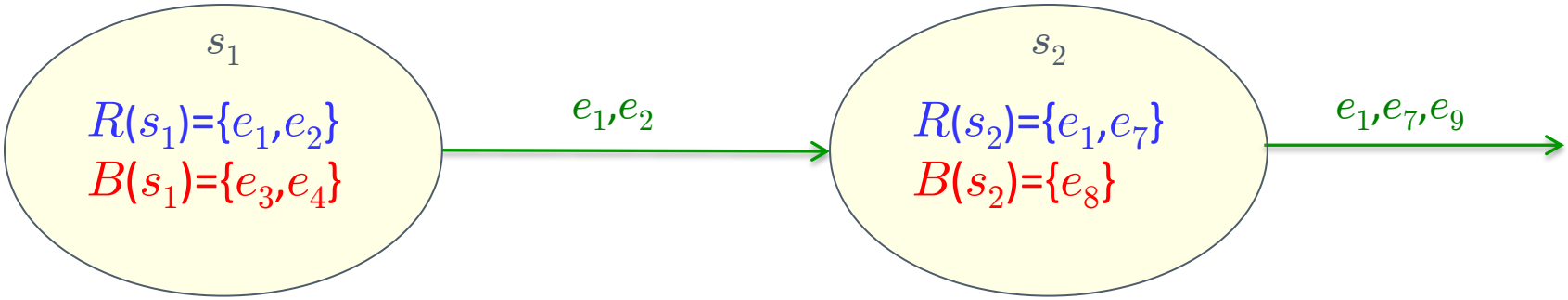
> a transition system $< S, E, \rightarrow, init >$

+ The transition system models the **waited for** events in each state

> for each state s:

+ a set $R(s)$ models the **requested** events

+ a set $B(s)$ models the **blocked** events



The runs of a set of b-threads

Composition of the b-threads $\{ < S_i, E_i, \rightarrow_i, init_i, R_i, B_i > : i=1,...,n \}$ is defined as a product transition system

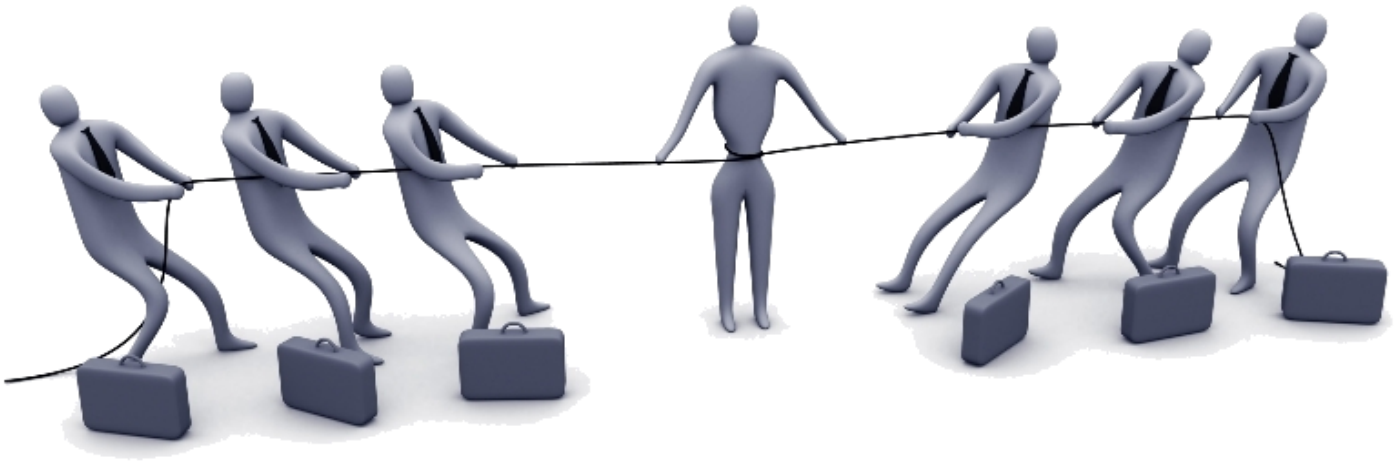
The composition contains the transition $\langle s_1, \ldots, s_n \rangle \xrightarrow{e} \langle s'_1, \ldots, s'_n \rangle$ if:

$$\underbrace{e \in \bigcup_{i=1}^{n} R_i(s_i)}_{e \text{ is requested}} \quad \bigwedge \quad \underbrace{e \notin \bigcup_{i=1}^{n} B_i(s_i).}_{e \text{ is not blocked}}$$

$$\bigwedge_{i=1}^{n} \Big(\underbrace{(e \in E_i \implies s_i \xrightarrow{e} s'_i)}_{\text{affected b-threads move}} \wedge \underbrace{(e \notin E_i \implies s_i = s'_i)}_{\text{unaffected b-threads don't move}} \Big)$$

Verification

» When each module is programmed separately, how do we avoid conflicts?



Dealing with conflicts and underspecification

"**Always stop** for pedestrians at crosswalks"

"**Never stop** when the vehicle behind you is too close"

Answer:

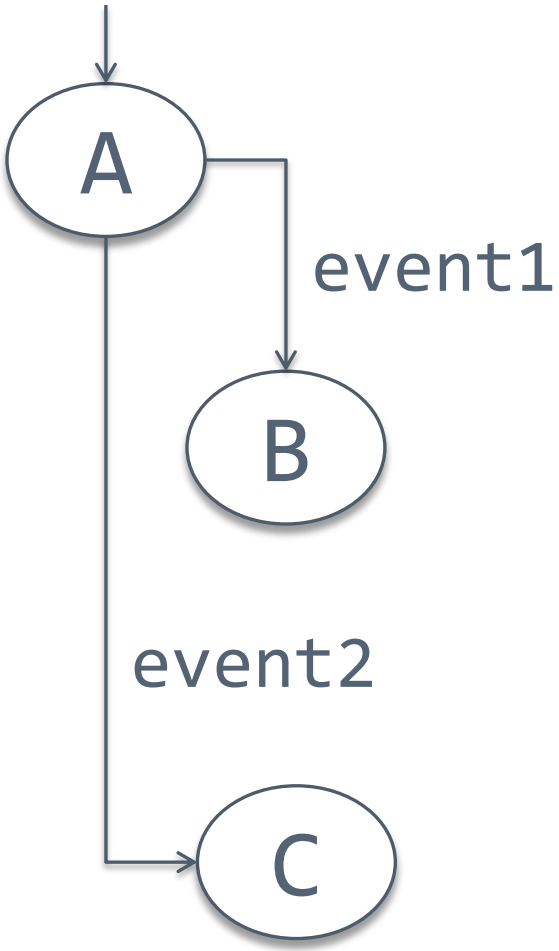
Model check the application + incremental development

[Harel, Lampert, M., Weiss, EMSOFT 2011]

Behavior Thread States
b-thread states at bSync

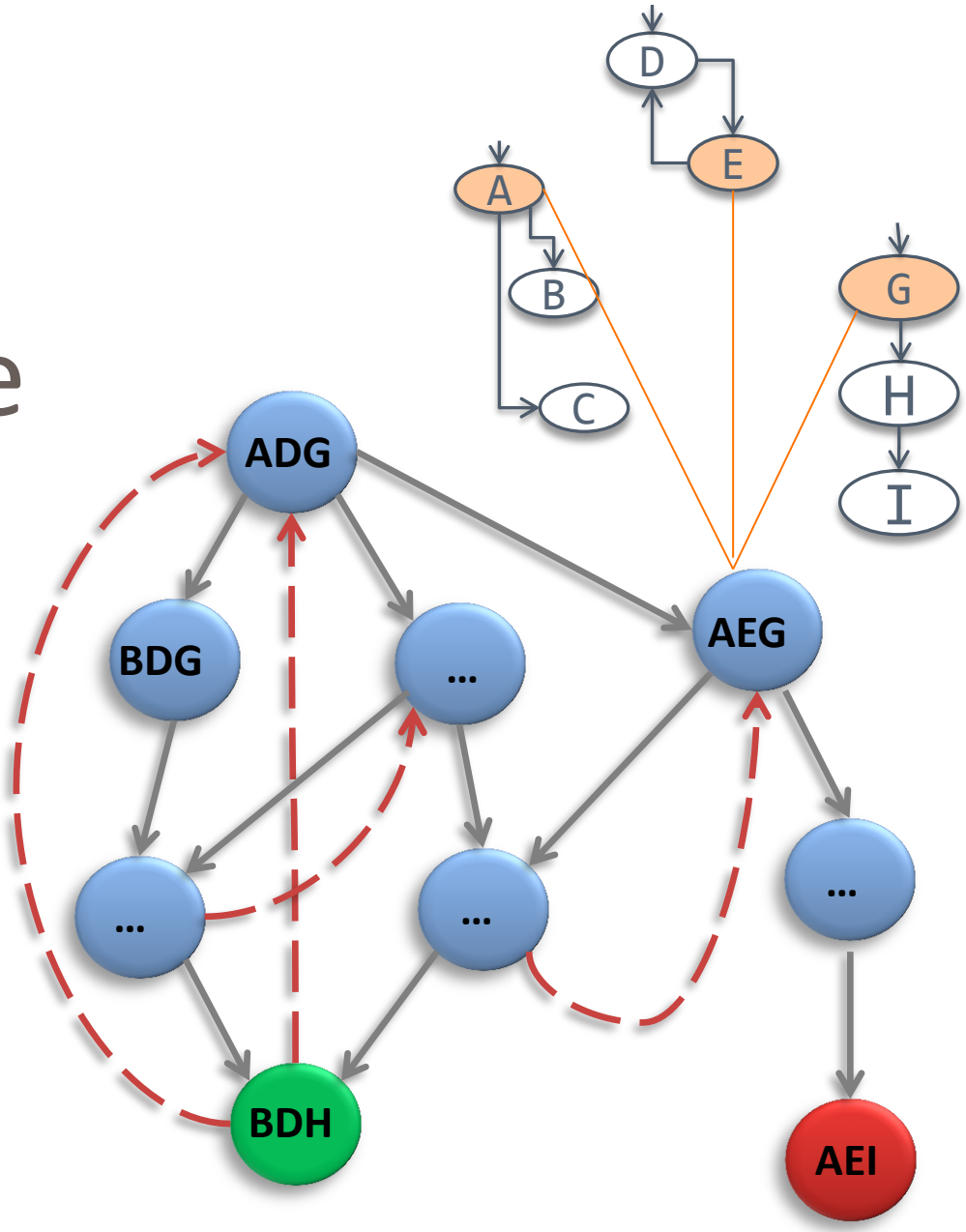
```
.
.
labelNextVerificationState( “A” );
bSync( … );
if( lastEvent == event1 ) {
        .
        .
        .
        labelNextVerificationState( “B” );
        bSync( … );
}

if( lastEvent == event2 ) {
        .
        .
        .
        labelNextVerificationState( “C” );
        bSync( … );
}
```



Behavioral Program State Graph

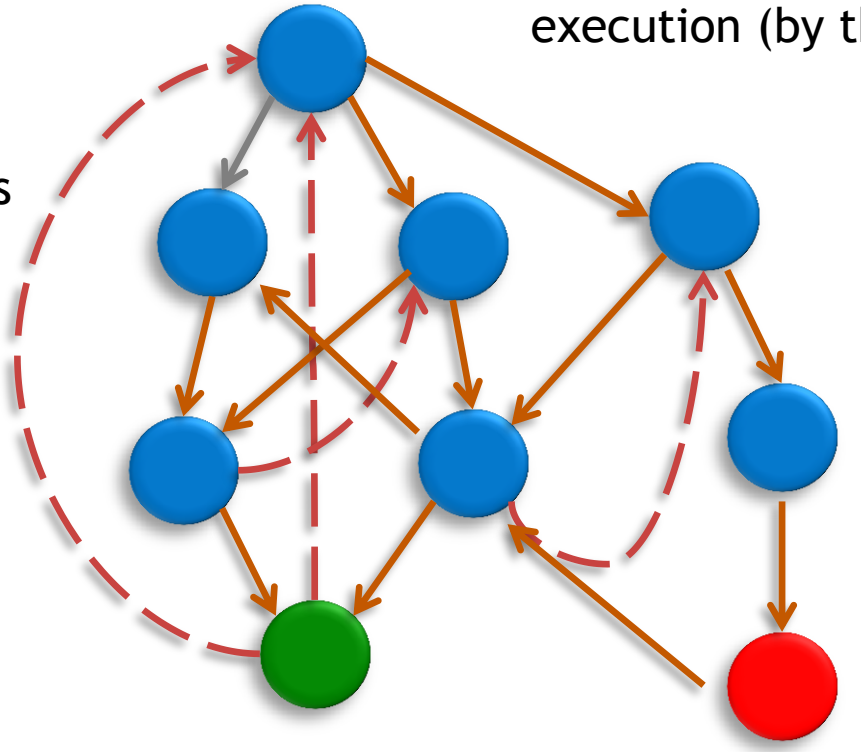
Program states are the Cartesian product of b-thread states



Model-checking behavioral programs "in-vivo"

Transition using standard execution (by the native JVM)

Backtrack using Apache `javaflow` continuations



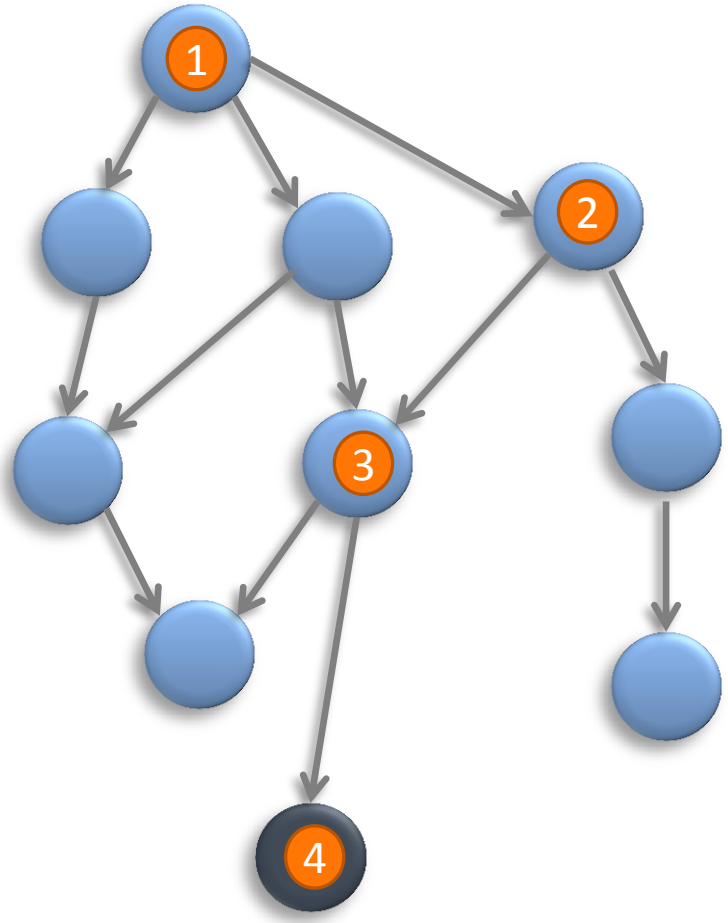
B-threads designate nondeterministic transitions

State matching and search pruning by b-threads

State tagging for **safety** and **liveness** properties by b-threads

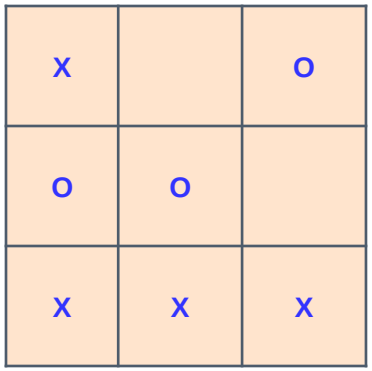
[Harel, Lampert, M., Weiss, EMSOFT 2011]

Counterexample: A path to a bad state



Model-checker-assisted development of Tic-Tac-Toe

- Initial Development:
  - `DetectXWin, DetectOWin, DetectDraw`
  - `EnforceTurns`
  - `DefaultMoves`
  - `XAllMoves`
- Modify b-threads to prune search / mark bad states
- Model Check → Counterexample → Add b-thread / change priority:
  - `PreventThirdX`
  - `PreventXFork`
  - `PreventAnotherXFork`
  - `AddThirdO`
  - `PreventYetAnotherXFork`



Counterexamples as scenarios

» Let $c=e_1, \ldots, e_m, \ldots, e_n$ be a counterexample

» Can generalize and code new b-threads or,

» Using counterexample in a patch behavior. E.g.,

> Let e_m be the last event requested by the system

+ Wait for $e_1, \ldots, e_{m-1}$

+ Block e_m

> Other b-threads will take care of the right action, “the detour”.

> Model-check again

Compositional Verification

- Design the program modules (threads)
- Formulate module properties in a theorem prover (Z3)
- Verify composite system with Z3 based on:
  - Module properties
  - BP composition properties (defined once)
  - **Conclude: system is correct if module properties hold**
- Implement the modules (say, in Java)
- Verify the Java modules directly - using a BP model checker

- Benefits:
  - This process guarantees correct system behavior
  - Design bugs detected before implementation
  - No explicit verification of the system

Compositional Verification Example : Counting

- Events: E_0, E_1
- We want to generate the runs where
  - E_1 can appear everywhere
  - E_0 can appear only at indices divisible by $3 \times 7 = 21$
  - $\left((E_1)^{20}\ (E_1 + E_0)\right)^{\omega}$

Designing the Threads

- B-thread *gen:*
  - **Requests events $\{E_0, E_1\}$** all the time **(1 state)**
- B-thread 1:
  - **Blocks event $\{E_0\}$** if index is not divisible by 3 **(3 states)**
- B-thread 2:
  - **Blocks event $\{E_0\}$** if index is not divisible by 7 **(7 states)**

- E_1 is enabled at every index
- E_0 is enabled only at indices divisible by 21

- The entire system has 21 states
  - Explicit model checking requires visiting all of them

Formulating Properties in Z3

$$\forall t, e: ((t\%3 \neq 0) \Leftrightarrow \texttt{blocked\_by}(E_0, t, BT_1)) \wedge \neg\texttt{blocked\_by}(E_1, t, BT_1) \wedge \neg\texttt{requested\_by}(e, t, BT_1)$$

$$\forall t, e: ((t\%7 \neq 0) \Leftrightarrow \texttt{blocked\_by}(E_0, t, BT_2)) \wedge \neg\texttt{blocked\_by}(E_1, t, BT_2) \wedge \neg\texttt{requested\_by}(e, t, BT_2)$$

$$\forall t, e: \texttt{requested\_by}(e, t, BT_{gen}) \wedge \neg\texttt{blocked\_by}(e, t, BT_{gen})$$

- Model-check each module separately to prove these properties
- Only explore $1 + 3 + 7 = 11$ states

BP Composition Properties

- Formulating the BP properties in Z3
  - Part of the framework, works for all applications

```
requested = Function('requested', Event,
                     Time, BoolSort())

blocked   = Function('blocked', Event,
                     Time, BoolSort())

trace     = Function('trace', Time, Event)
```

$$\forall e, t: \text{trace}(t) = e \Rightarrow \text{requested}(e,t) \wedge \neg \text{blocked}(e,t)$$

Proving Desired Behavior

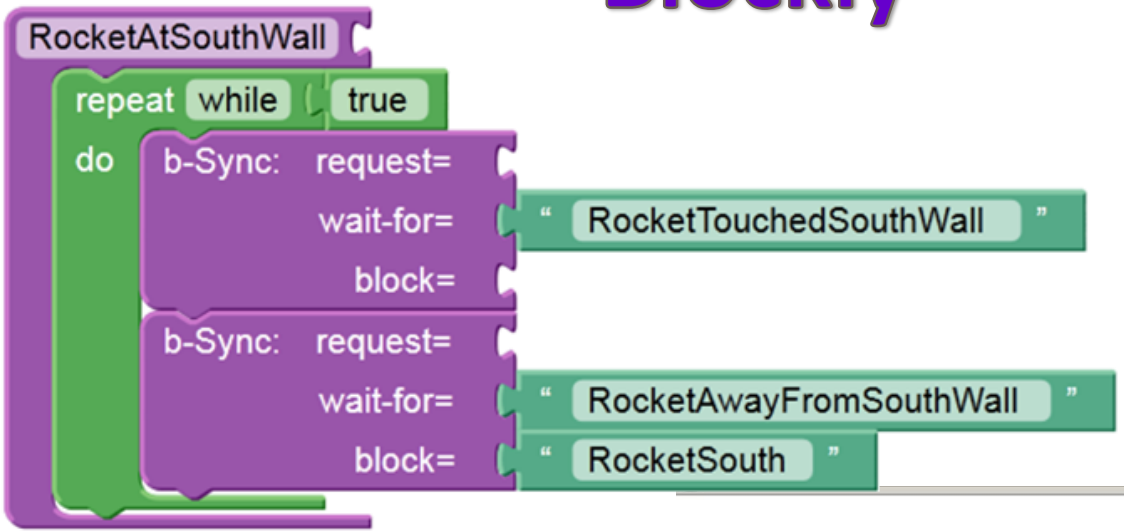
» Finally, have Z3 prove the desired property

$$\forall t:\ \texttt{requested}(E_0,t) \wedge \neg\texttt{blocked}(E_0,t) \Leftrightarrow t\%21==0$$
$$\forall t:\ \texttt{requested}(E_1,t) \wedge \neg\texttt{blocked}(E_1,t)$$

» Z3 answers: **the property holds!**

Programming a Game in Blockly

Blockly


RocketAtSouthWall
repeat while true
do
b-Sync: request=
wait-for= " RocketTouchedSouthWall "
block=
b-Sync: request=
wait-for= " RocketAwayFromSouthWall "
block= " RocketSouth "


^ North
< West
UP
East >
StartGame
Status: Playing
TIME=0
v South

A program playing Tic-Tac-Toe

```
EnforceTurns() {
   forever {
      bSync(request=none, wait-for=XMove, block=OMove);
      bSync(request=none, wait-for=OMove, block=XMove);
   }
}
```

⋮

Rules of the game

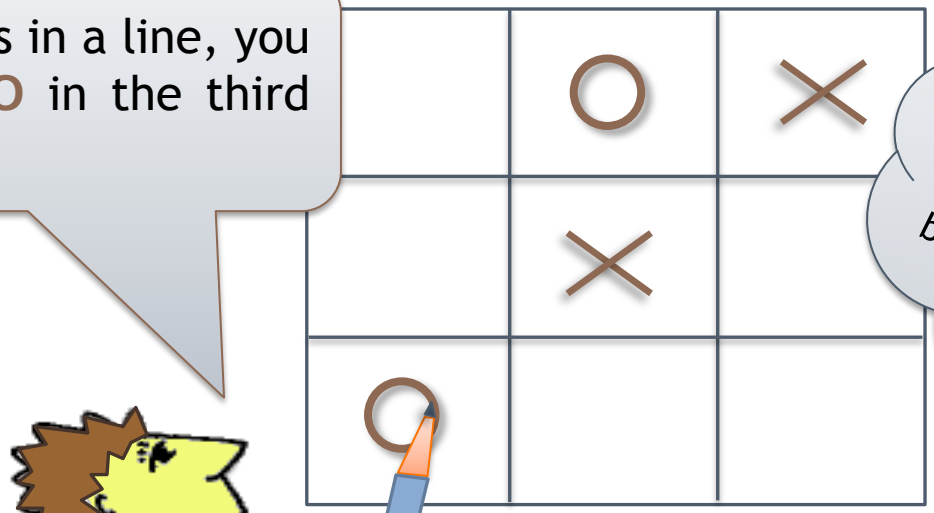
```
PreventThirdX() {
      bSync(request=none, wait-for=X⟨1,3⟩, block=none);
      bSync(request=none, wait-for=X⟨2,2⟩, block=none);
      bSync(request=O⟨3,1⟩, wait-for=none, block=none);
}
```

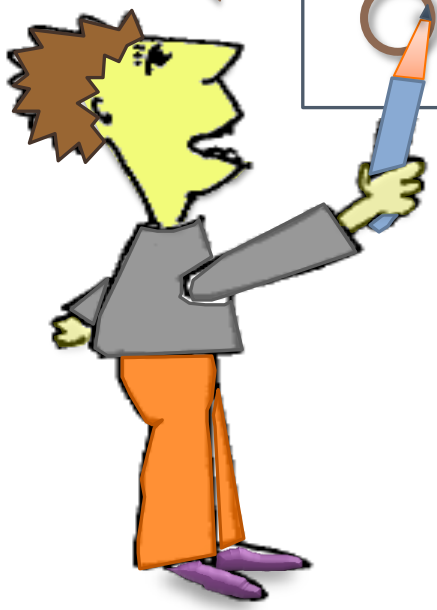
⋮

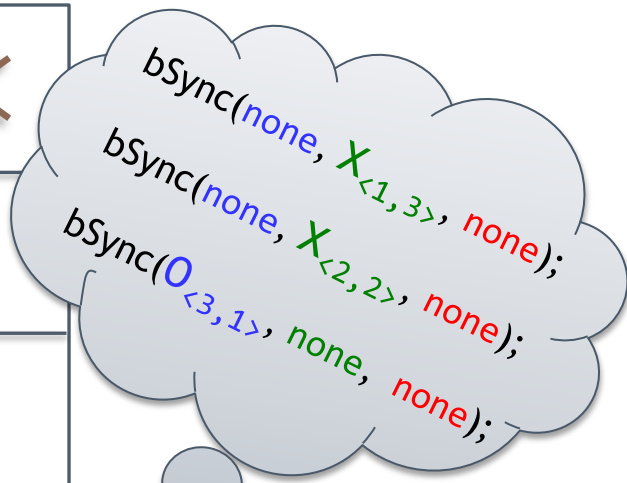
Strategies

Game Example:
Alignment of code modules with requirements

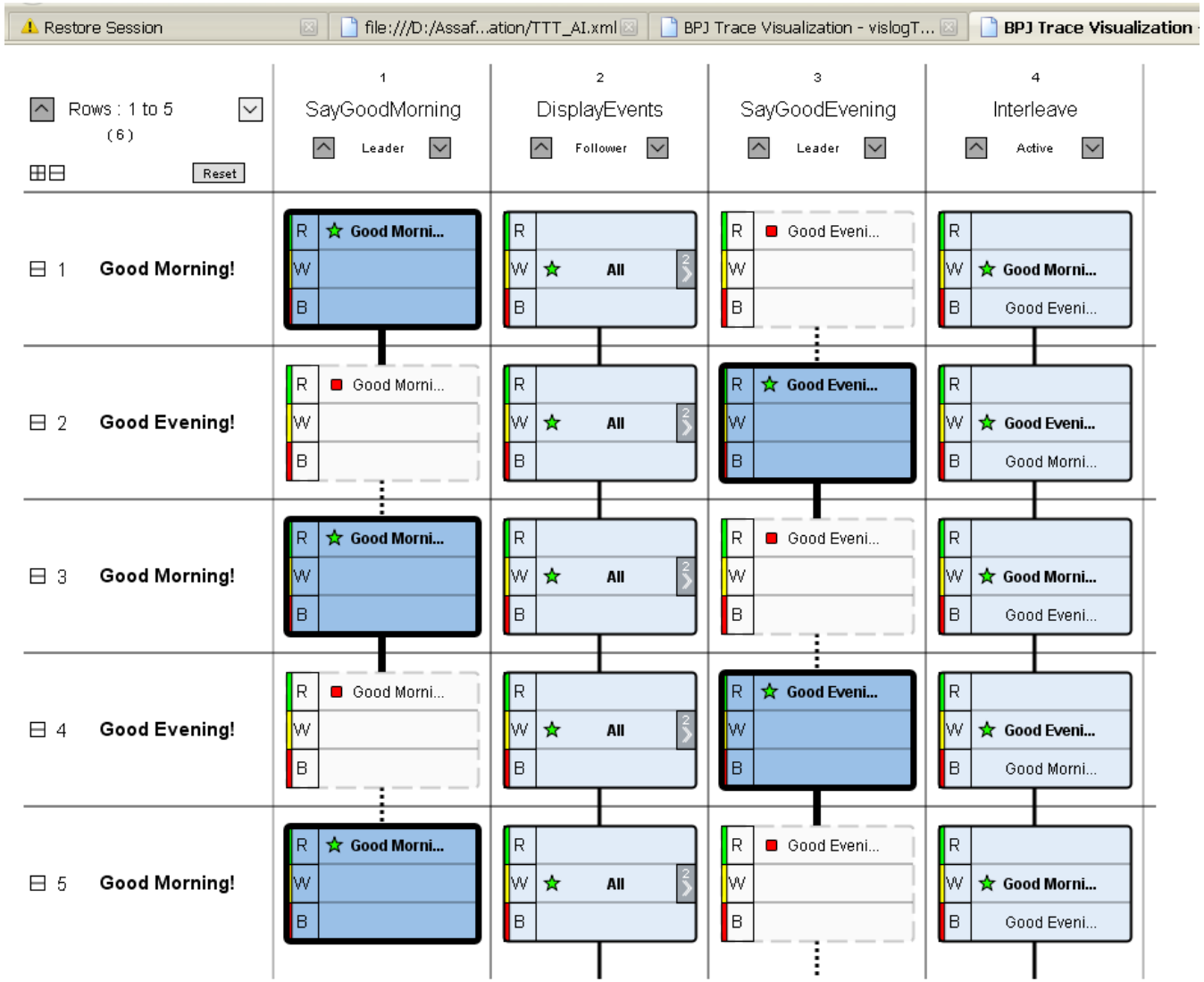
When I put two Xs in a line, you need to put an O in the third square







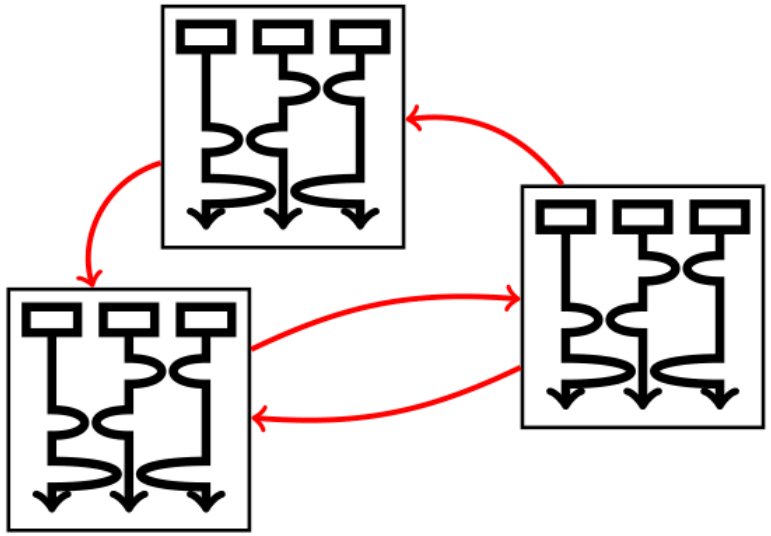
Visualization



Remote Events – Local Behavior

Real-life behavioral applications require distributed execution

- Asynchronous communication between nodes
- Synchronous collaboration inside nodes
- Each node has scenarios for handling remote events



Incremental development in Java with BPJ

Req. 3.1

```
class AddHotFiveTimes extends BThread {
    public void runBThread() {
        for (int i=1; i<=5; i++) {
            bSync(addHot, none, none);
        }
    }
}
```

Behavior Threads

Req. 5.2.9

```
class AddColdFiveTimes BThread {
    public void runBThread() {
        for (int i=1; i<=5; i++) {
            bSync(addCold, none, none);
        }
    }
}
```

Patch 7.1

```
class Interleave extends BThread {
    public void runBThread() {
        while (true) {
            bSync(none, addHot, addCold);
            bSync(none, addCold, addHot);
        }
    }
}
```

Towards incremental development

Need to accommodate a cross-cutting requirement? **Add a module**

Need to refine an inter-object scenario? **Add a module**

Need to remove a behavior? **Add a module**

. . . ? **Add a module**

Thank You!