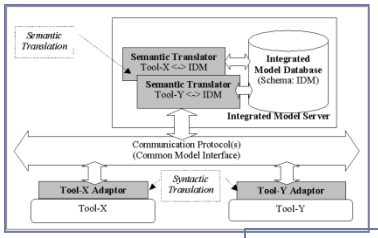
VANDERBILT UNIVERSITY
INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS

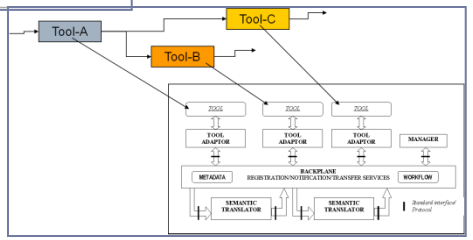
Integration and Verification of Models with Heterogeneous Semantics

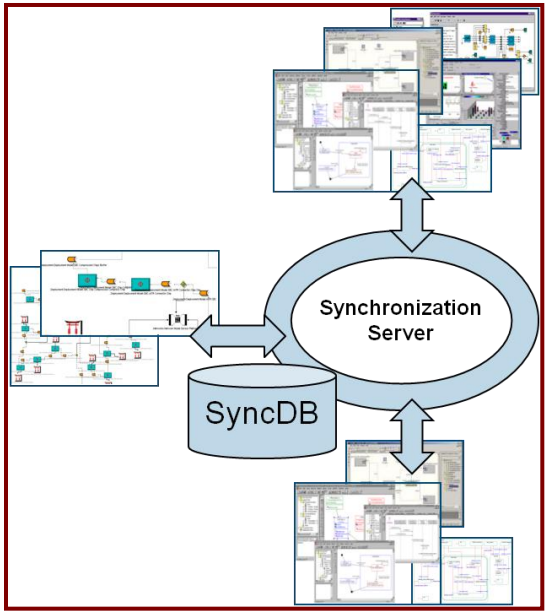
Gabor Karsai
Vanderbilt University/ISIS

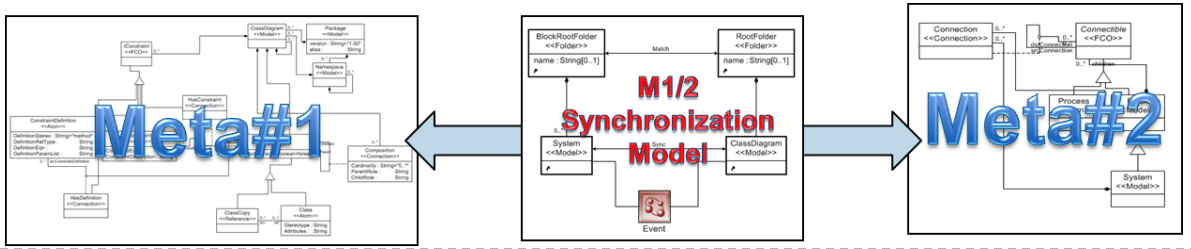
Proposal Topics
Composition Framework for Tool Integration

- Tool Integration Design Patterns
  - 'Star' – Common Integrated Model
  - 'Workflow' – Translators
  - → Do not fit collaborative work well…
- Distributed collaborative work needs…
  - → Model Synchronization
    - How to model dependencies among models?
    - How to support asynchronous work?
    - How to manage versions?
    - How to propagate changes?



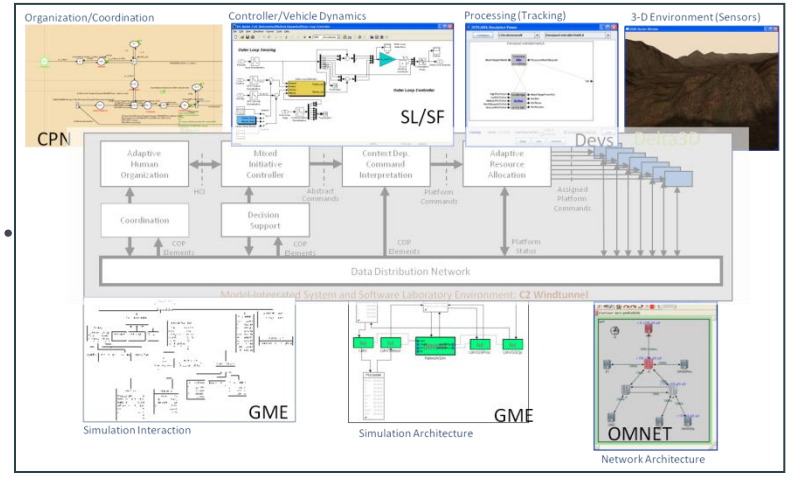




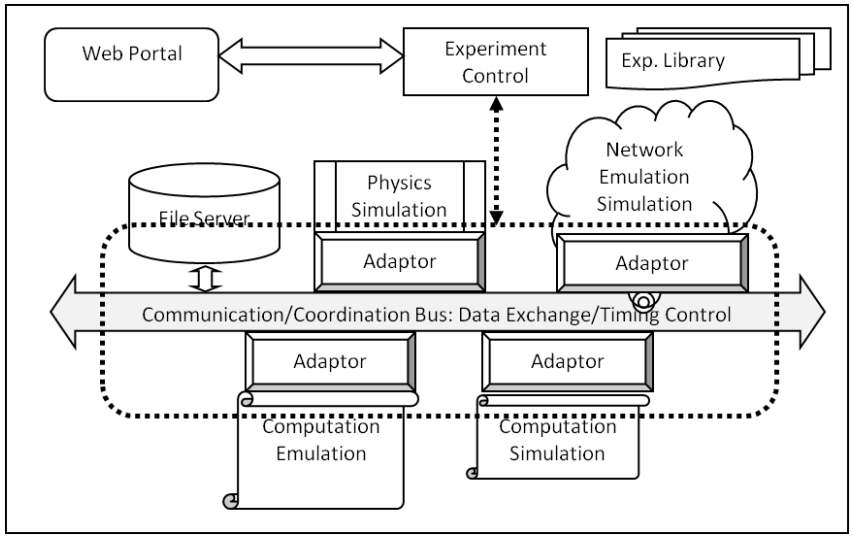


Proposal Topics
Multi-model Simulation Integration

- C2 Wind-Tunnel: MSI for C2
  - Integration Model: HLA federates
  - Not suitable for real-time systems...

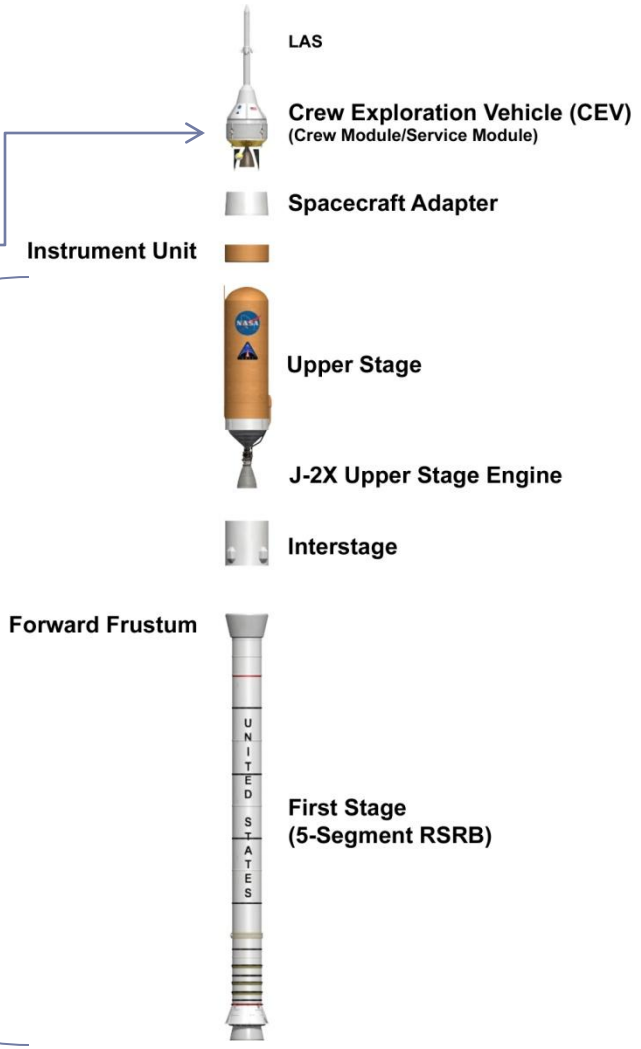


- Virtual Prototyping for CPS:
  - Fine-grain time control
  - Models for platforms
    - Processors, networks, middleware
  - Integration of emulators
    - Cycle-accurate timing
  - Integration of physics models
    - Multi-scale timing



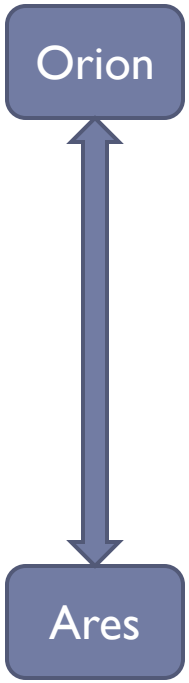
An example:
A Cyber-Physical System Integration Problem

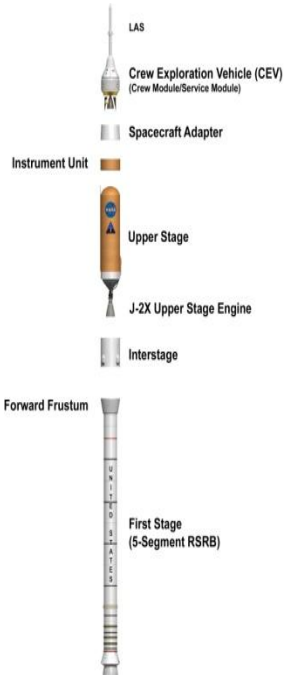
- The next Manned Spaceflight System:
  - Orion: Crew Exploration Vehicle
  - Ares: Booster



A Cyber-Physical System Integration Problem

- Orion: GNC is in Simulink/Stateflow
  - Stateflow: A graphical modeling language based on Statecharts (Harel, 1988)
- Network ???
- Ares: GNC is in UML / Rhapsody
  - UML/State Machines: A graphical modeling language based on Statecharts (Harel, 1988)





Some small *problems…*

1. *Semantics(Stateflow) ≠ semantics(UML State Machines)*
2. *Message sequencing on the network is not defined*

Challenge:
How to analyze/verify such systems?

Problem #1: Semantic variants

- Statecharts: > 20 variants (von der Beek, 1994)
- Semantics described formally in papers
  - Stateflow variant (e.g. Rushby, 2004)
- Semantics described in documents
  - Mathworks Stateflow documentation
  - UML State Machines (OMG UML Standard)
- Comparing semantics
  - Composable Semantics (Atlee , 2002)
  - Structured Operational Semantics (Whalen, 2010)

Dynamic Semantics of DSMLs

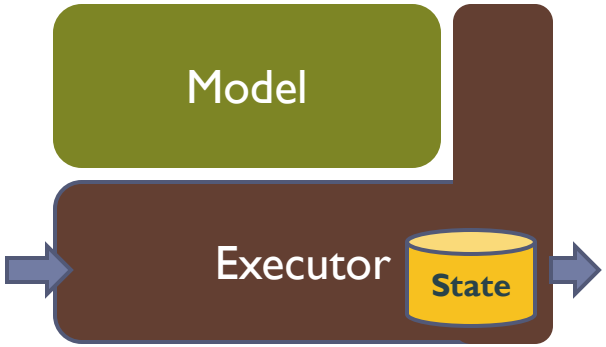
- Pragmatic needs:
  - Executable, understandable semantics so one can 'simulate' model behavior → Execution-based
  - Models are often transformed into artifacts (even if they are not executable) → Translation-based
  - Property checking on models → Evaluation-based
- The reality:
  - Rapid prototyping of semantics is important
  - Property checks can often be done by well-formedness rules, decision procedures can be 'programmed' w.r.t the metamodel
  - Need: reusable 'semantic platform'

Dynamic Semantics of DSMLs

- The 'semantic platform' – the realization of the semantic domain:
  - ASML
    - MSR's implementation of Gurevich's Abstract State Machine
      - Abstract state: the state of all the variables in the program
      - Actions: updates on the abstract state
    - Like a OO programming language
    - Mid-level, tools for simulating and exploration (SpecExplorer)
  - "Semantic units"
    - High-level, reusable packages representing typical semantic concepts
    - Example: finite transition system (in ASML)
    - Assumption: variant DSMLs can be translated into a common S/U
  - Java/JVM: low level, efficient, tools for model checking
    - JPF: Java Path Finder

Semantics via Model Execution

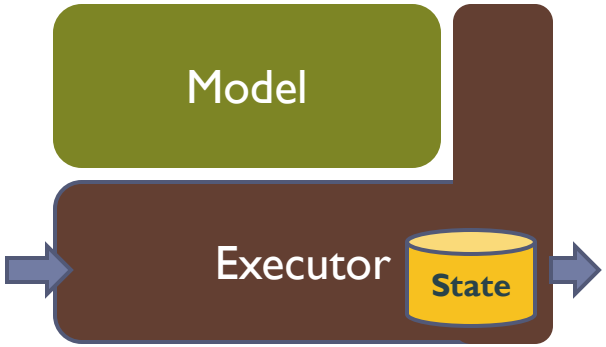
- Executor: A generic 'model interpreter' whose 'program' is the model
- A formally specified executor *defines* the semantics



- The 'Model' is just static, constant data for the executor
- Formally specify executor:
  - Use a (executable) specification language – ASML
  - Use a (conventional) implementation language ?!

Example
Model Executor

- Specifying the Executor using an executable spec (a.k.a. 'program')
  - Everybody understands it
  - Everybody can reason about it
  - Allows quick prototyping
  - Program verification, debugging, testing tools can be applied



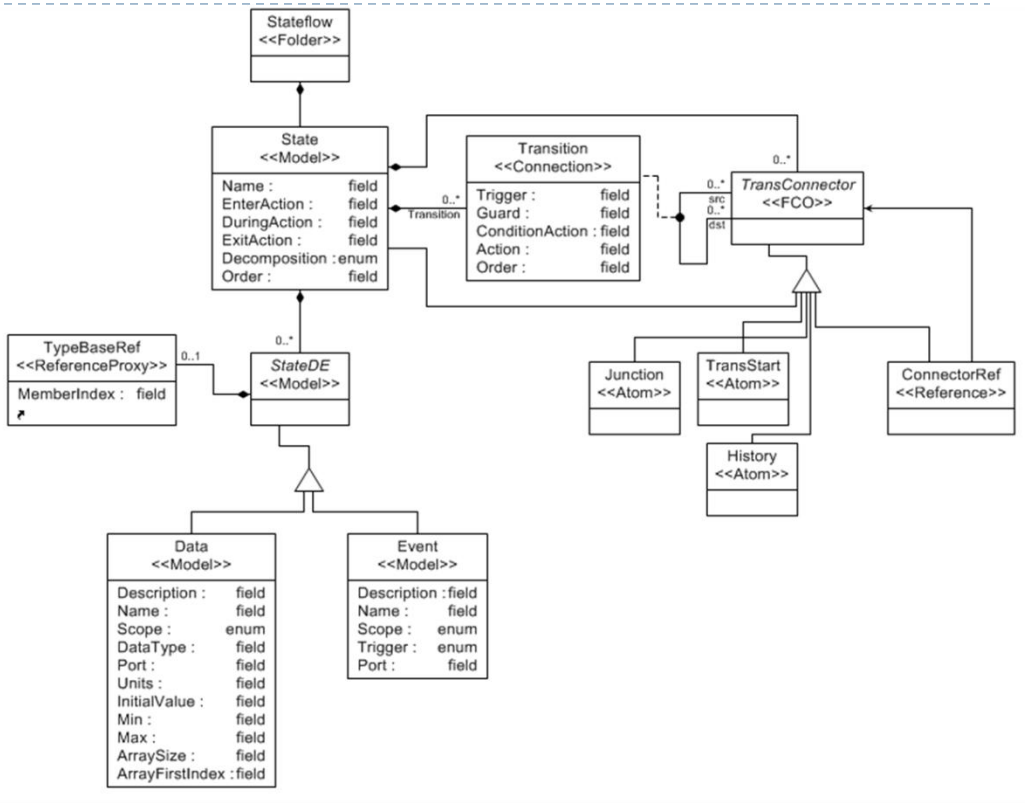
- Downside:
  - Reasoning about (general) programs is more difficult than reasoning about models
  - Inefficient in verification (too many states)
  - Non-determinism is a problem

Example
Semantics for Statechart variants

- Many Statechart variants – our examples:
  - UML State Machines
  - Matlab Stateflow
- Specification:
  - Natural language (< 100 pg, each)
  - For some subsets: formal spec (in papers)
- Need:
  - Executable specification for the semantics of both
  - Goal: to apply verification tools (JPF) to the models

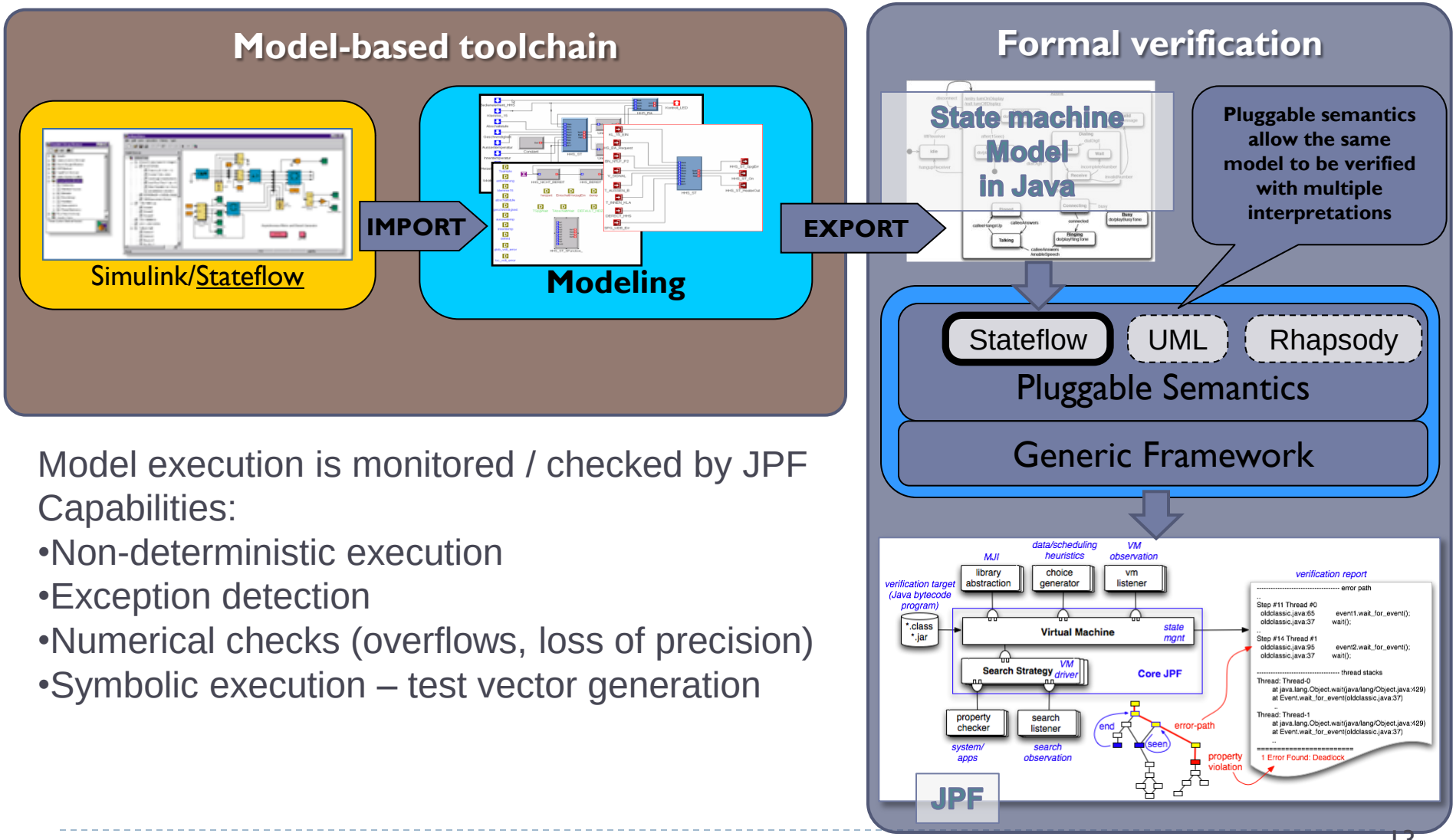
Preliminary result: Semantics for Statechart variants

- Executable semantics
  - Common Metamodel
    - Abstract syntax



  - Executor (Java)
    - Data model: ~ 600 lines
    - Common interpreter code: ~ 250 lines
    - Stateflow: ~600 lines
    - UML State Machine: ~ 400 lines

Example: (Stateflow)
How is the executable semantics used?



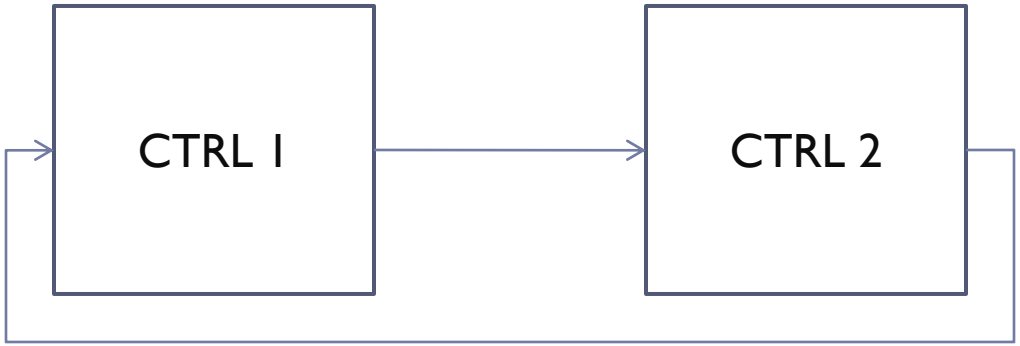
Model execution is monitored / checked by JPF

Capabilities:

- Non-deterministic execution
- Exception detection
- Numerical checks (overflows, loss of precision)
- Symbolic execution – test vector generation

Problem #2: Concurrency

- Two systems connected through a network
  - How do we model, analyze these?
- Try Simulink:

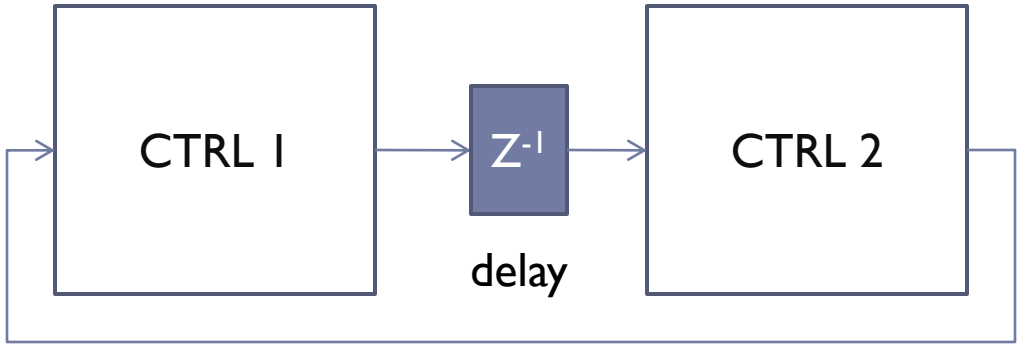


What it means:
at time t: ctrl1.in = ctrl2.out AND ctrl1.out = ctrl2.in

But: it is an algebraic loop – Simulink *cannot* simulate!

Problem #2: Concurrency

- Two systems, connected - fixed

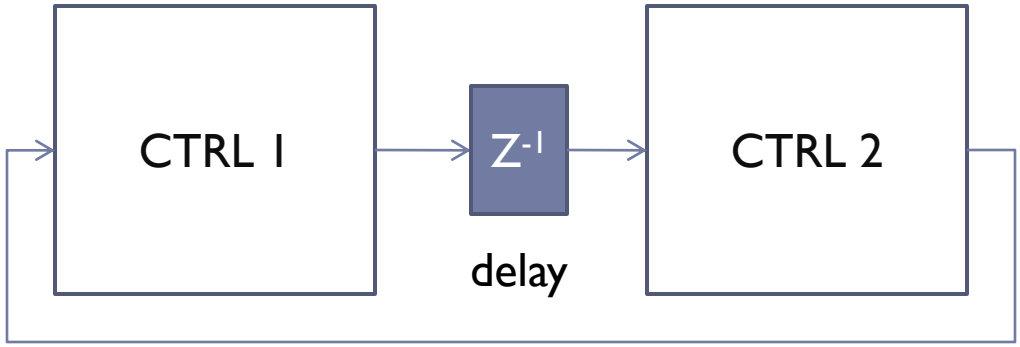


z^{-1} : '1-step' delay : output is delayed a non-zero time-step (*delta*).
Has an initial value: at t=0 its output is defined.

What it means:
at time t: ctrl1.out = delay.in AND delay.out = ctrl2.in AND ctrl2.out = ctrl1.in AND delay.out = delay.in @ t - delta

Now Simulink can simulate!

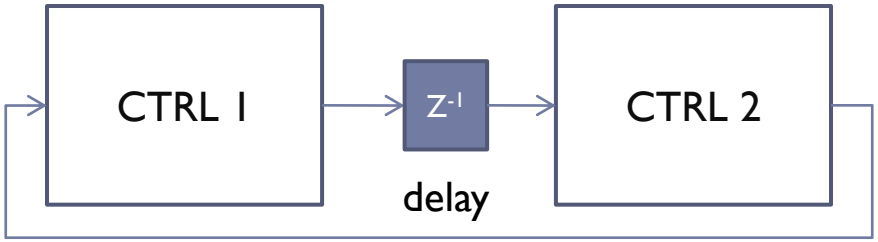
Concurrency



Simulation:
- Simulink applies the 12'o clock rule + checks what has data
- t = 0: CTRL2 ; CTRL1
  t = delta : CTRL2; CTRL1
  t = 2*delta: CTRL2; CTRL1 ….
I.e: { CTRL2 ; delta ; CTRL1 }+ forever --- Is this real???

Two problems:
1. CTRL1 and CTRL2 takes some time to execute, which may be different (e.g. WCET(CTRL1) and WCET(CTRL2)) → trivial to fix
2. In real life:
- CTRL1/2 are on different processors that communicate via a network
- CTRL1/2 run as independent threads that communicate asynchronously

Concurrency



Simulation implicitly does this:
{ CTRL2.recv(); CTRL2.run(); CTRL2.send(); → CTRL1.recv(); CTRL1.run(); CTRL1.send(); }+

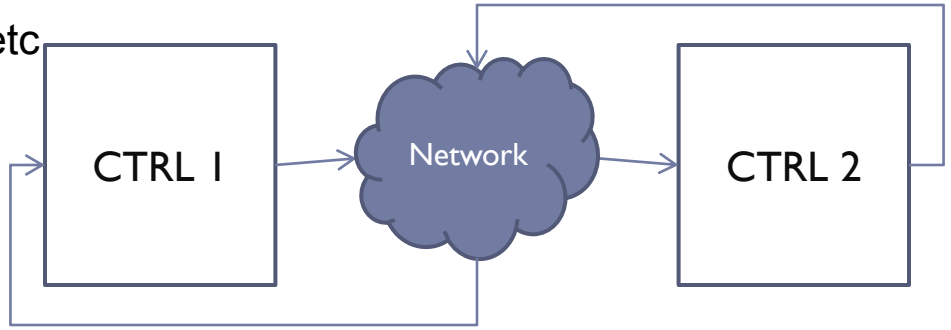
System does this – two independent, asynchronous processes:

P1: { CTRL1.recv(); CTRL1.run(); CTRL1.send(); }+

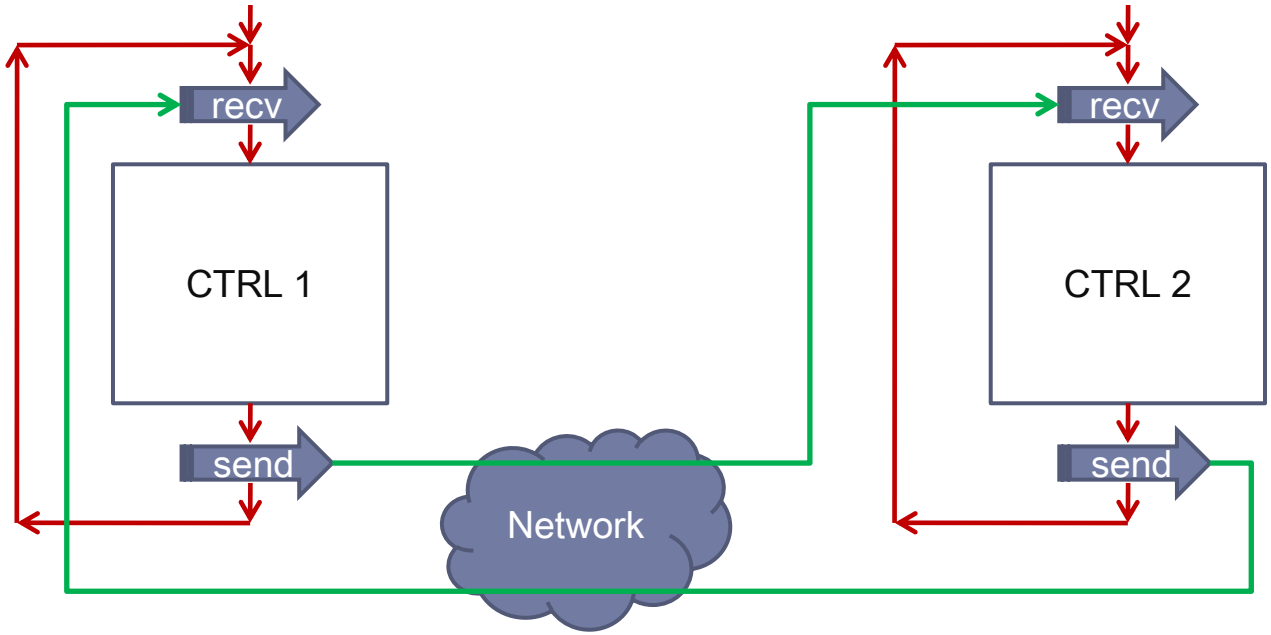
P2: { CTRL2.recv(); CTRL2.run(); CTRL2.send(); }+

If receive blocks, it will deadlock…

And the network?
- Communication buffers, queues, delays, etc
- Protocols that transfer the messages



Control flows – data flows



Some variants: (they need to be modeled)
1. First occurrence of one of the receive()-s does not wait, it returns a 'default' value → Still lock-step execution
2. Receive() does not block, controllers always work from the last input → receive() gets its data from the network buffer (de-coupled execution)
3. Receive delivers *data*, which can trigger an *event* that will trigger the controller → the presence of an event depends on the data value

Modeling Concurrency + Communications

Background: Calculus of Communicating Systems (CCS) by Milner – a process algebra

Highlights:

Agents: active entities

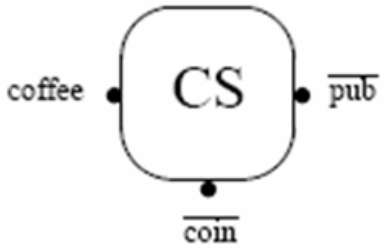
Actions: synchronization points:

a : 'receive on port a'

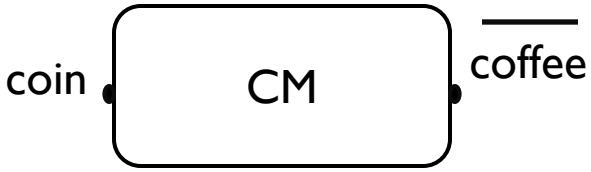
$\overline{a}$: 'send on port a'

CCS

- Example 1: An agent (process) CS (for Computer Scientist) with input: coffee and outputs: coin and pub.



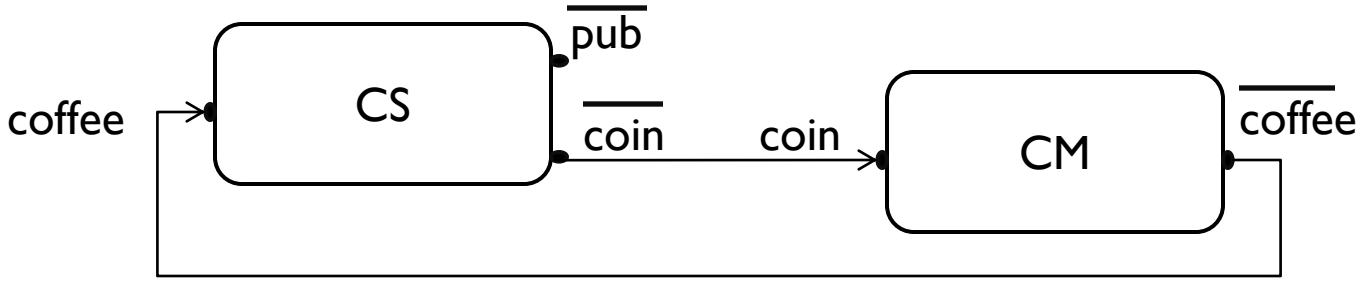
- Example 2: Coffee machine



- Behavior $\overline{\quad\quad}$
  CM = coin.$\overline{coffee}$.*CM*
  Receive a coin, send coffee..

CCS: Composition

- CS | CM



- CS = coffee.$\overline{pub}$.CS + $\overline{coin}$.CS
- CM= coin.$\overline{coffee}$.CM

  CS | CM: infinite producer of pubs ☺

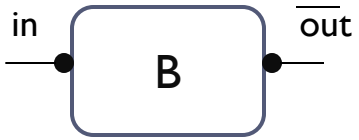
CCS

- Models processes and their potential communications: synchronization points and their temporal sequencing
- Operators:
  - prefixing: in.*P*, $\overline{\text{out}}$.*Q*
  - alternative actions: in1.$\overline{\text{out1}}$.*P* + in2.$\overline{\text{out2}}$.*P*
  - composition: P | Q: wire up compatible ports
  - much else (left out)

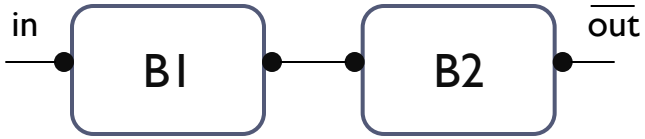
CCS - Examples

- Single buffer

  $B = in(X).\overline{out}(X).B$



- Buffer with capacity = 2



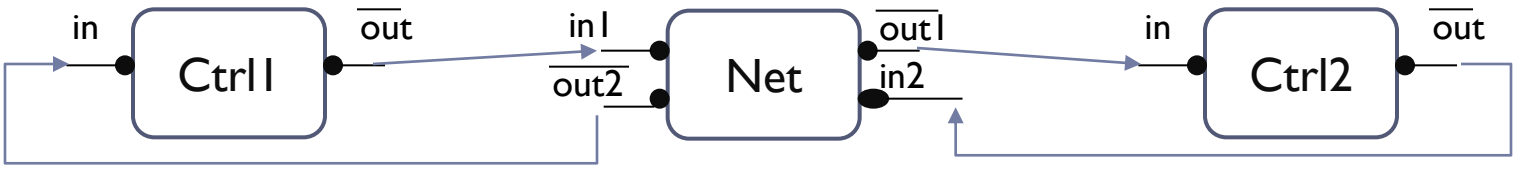
- "Guaranteed delivery"

  $D = in(X).\overline{out}(X).ackout(X).\overline{ackin}(X).D$

- Two-way buffer

  $B = in1(X).\overline{out1}(X).B + in2(X).\overline{out2}(X).B$

Our example: Two controllers + network



A Ctrl_ process receives on its in, computes, sends on its out, then repeats

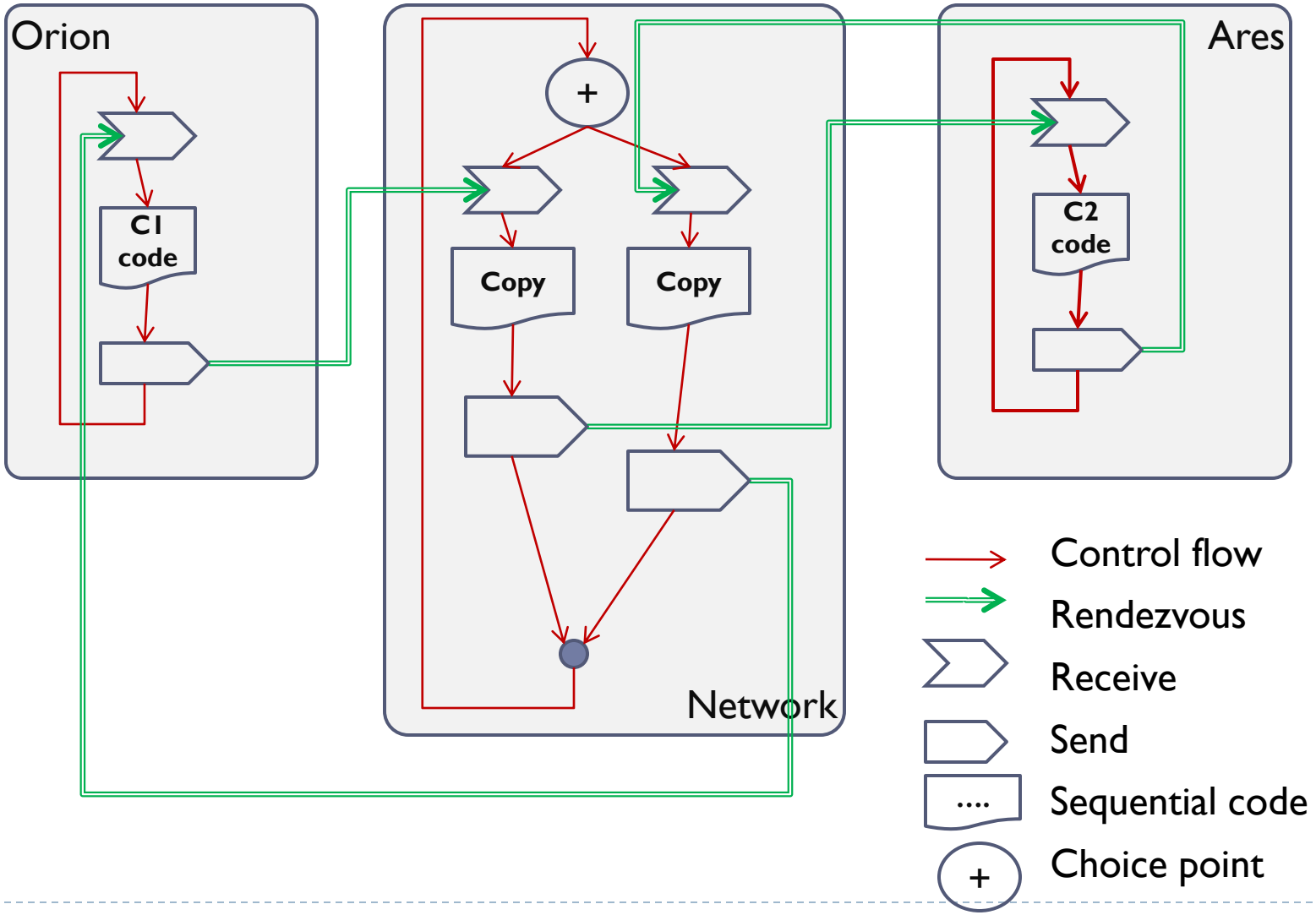
The network connection either receives on in1, then copies, then sends on out1, or receives on in2, then copies, then sends on out2; and then repeats

Of course this deadlocks immediately, but this is one thing we would like to catch…

Modeling with CCS

- Both Simulink/Stateflow and UML State Machines are translated into sequential code blocks.
- Need to model:
  - How the sequential code blocks are embedded into processes (that have their own execution threads)
  - How the network communication / message exchanges are sequenced
- Interaction point: rendezvous ($\overline{out}$ → in)

Modeling the problem



Executable semantics

- Each loop is an independent Java thread
- *Rendezvous point*: whoever arrives earlier, waits for the other
  - Has two parties: sender and receiver
- *Non-deterministic choice* point: multiple rendezvous are possible (both sender and receiver)
  - Whichever succeeds (= the other party arrives) first, will proceed

- The above scheme can be implemented using a simple Java library, code can be generated from the model, and the result Java model of the system analyzed (using, e.g. JPF)

Research challenges

- Core integration problem:

  How to check the correctness of the integration in advance?

- Modeling language and tool for timed concurrent systems
- Models for typical communication / concurrency patterns
- Model generators:
  - Transforming the models into concurrent Java code
  - Transforming the models into other, analyzable formalisms
- Analysis techniques
  - Java Path Finder
  - SMV? Promela/SPIN? etc.
- Evaluation on examples