# Introduction to SHIFT and SmartAHS

## Late 1996

Aleks Gollu





## Overview

- History of project
- Concepts in SHIFT
- SHIFT language overview
- SHIFT example -- moving particles
- SHIFT mathematical model
- SHIFT example -- SmartAHS
- SHIFT development methodology
- Remaining work

http://www.path.berkeley.edu/shift





## Project History





### The Needs

- Provide tools for application domains with the following characteristics
  - The behavior of objects in the system have both continuous and discrete event components -- hybrid systems;
  - The systems consist of heterogeneous set of interacting objects where models of individual objects are known and the goal is the study of the **emergent** behavior resulting from their interaction;
  - A static block diagram representation is not sufficient to specify all data dependencies among objects since the sets of objects that interact vary over time.

## Application Domains

- Highway Planning
  - Automated Vahicles, Automated Highways
- Air Traffic Management
- Underwater operations
  - Automated Submarines
- Material handling systems
  - Copiers, Color-Picture processors
- Mobile robot operations
- Manufactutring floors





## Existing Approaches/Tools

- Matlab, Mathematica, Matrix -X
  - No concept of “instantiating” an equation prototype
- Block diagram based, domain specific frameworks
  - Can’t add or reconnect blocks at run-time
- Class library based frameworks
  - Too many “how-to-use” rules that can’t be enforced
  - Artificial syntax and semantics

- Pieces of the problem is solved, but neither approach solves all requirements!





## What SHIFT Offers

- A programming language with simulation semantics and explicit domain specific syntax
- A mathematical model that defines the formal semanitcs
- Architected to have GUI, Command line, API interfaces
- Architected to facilitate porting to parallel processors
- Several application frameworks under development
- Available through
  - http://www.path.berkeley.edu/shift





## SHIFT Concepts

## Concepts in SHIFT

- SHIFT is a programming language intended for hybrid system specification and simulation
- SHIFT provides syntax for the specification of
  - objects with *discrete* and *continuous* behavior
  - objects have *discrete states* and *continuous variables*
  - each discrete state has a set of *differential equations* governing the time evolution of continuous variables
  - differential equations among object can be *linked*
  - *discrete transitions* (with events, guards, resets) govern the discrete state a given object is in
  - transitions among objects can be *synchronized*





## Class Based Modeling

- "type"s are used to describe prototypical behavior of objects
- "component"s of a given type are instantiated to populate the "world"

**Type Specifications**

```
type Vehicle {
  output continuous number x,
v;
  ...

}
```

**Components in the World**

```
Vehicle 0
x = 23.12
v = 20.01
...
```

```
Vehicle 1
x = 47.67
v = 20.12
...
```





## Links

- Links (references) among objects
- In types they are variables, in components they are component ids

**Type Specifications**

```
type Vehicle {
  ...
    Vehicle frontVehicle;
  ...

}
```

**Components in the World**

```
Vehicle 0
...
frontVehicle Vehicle 1
```

```
Vehicle 1
...
frontVehicle Vehicle 2
```





## Links, continued

- Links are *dynamic*

```
Vehicle 0
frontVehicle Vehicle 1
```

```
Vehicle 1
frontVehicle Vehicle 2
```

```
Vehicle 3
frontVehicle {}
```

**World at t = t1**

**World at t = t2**

```
Vehicle 0
frontVehicle Vehicle 2
```

```
Vehicle 3
frontVehicle Vehicle 1
```

```
Vehicle 1
frontVehicle {}
```

## Event Synchronization

- Types have *local events*
- A local event can be referenced by other types as an *external event*
- A local event is synchronized with all external events referencing it resulting in

**Type Specifications**

```
type Initiate {
  Response responder;
  ...
    join -> mergingI
      {responder:merge}
  ...}
```

```
type Response {
  ...
    lead -> mergingR
      {merge}
  ...}
```

**Components in the World**

```
Initiate 1
responder Response 1
join -> mergingI
  {Response1:merge}
```

```
Response 1
lead -> mergingR
  {Response1:merge}
```

a *collective transition* in the world



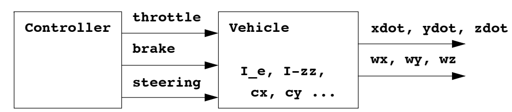


## Input, Output, State

- Distinction of *input*, *state*, and *output* variables
  - Each type can only read its inputs
  - Each type specifies the evolution of its state and output variables
  - Only the outputs of a type are available to other types
- Differential equation right-hand-sides can depend on outputs of other types

```
x' = f (x, v, x(frontVehicle), v(frontVehicle) )
```

- Differential equation right-hand-sides can depend on inputs, which are connected to other outputs




```
x' = f (I_e, I_zz, ..., throttle, steering, brake)
```





## The World

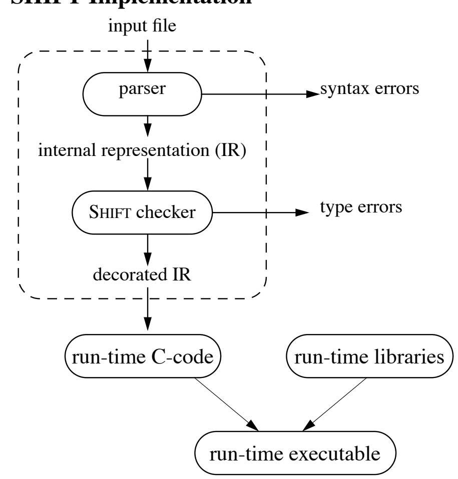
- The world state is given by
  - the type definitions in the world
  - the set of components in the world
  - the discrete mode of components
  - the (continuous) numeric variables of components
  - the link variables of components
- The World Evolution
  - As time passes the continuous numeric variables evolve according to the differential equations - continuous phase
  - During discrete transitions (in zero time) numeric and link variable values are reset, new components are created - collective transitions





## SHIFT Implementation

IR: a standardized data structure which contains all the information from the input file.



## Current Simulation Semantics

- We loop over continuous and discrete steps

- “continuous step”
    - 4th order Runge-Kutta integrating differential equations
    - fixed step size that can be changed for each simulation run
- “discrete step”
    - while a collective transition is possible execute it





## Language Overview





## Hybrid System TIF (SHIFT)

- Prototype-based Description

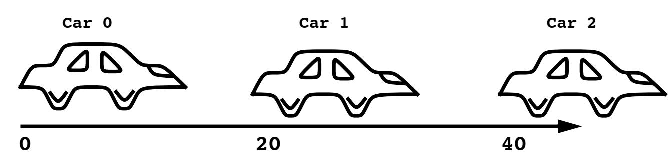
```
type Car {

  input ...what we feed to it
  output ...what we see on the outside
  state ...what's internal                   // data model
  discrete ... discrete modes
  export ... event labels

  flow ...
  transition ...        // continuous and discrete (called hybrid) evolution laws

  setup ...actions executed at creation time

}
```





## Example: populating a scenario

```
create(Car, x := 0, v := 0);
create(Car, x := 20, v := 0);
create(Car, x := 40, v := 0);
```




- This creates three **component**s (instances) of **type** Car with different intial conditions

## How to create a simulation

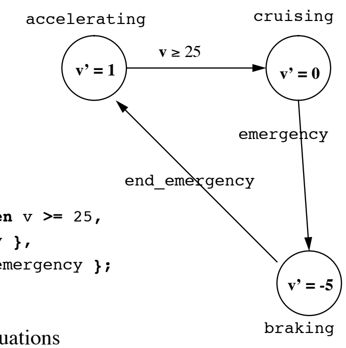
- An initial set of components are created and placed in the **world**
- The components evolve in two phases following the rules specified by their types
  - continuous phase
    - as time passes, components evolve according to the flow equations
  - discrete phase
    - time stops, components evolve according to discrete transitions, and take actions that
    - may create new components
    - and change interactions among components





## Example: toy car

```
type Car {
  state continuous number x, v;
  flow
    default {x′ = v, v′ = 0}
  discrete
    accelerating {v′ = 1},
    cruising,
    braking {v′ = -5};
  transition
    accelerating -> cruising {} when v >= 25,
    cruising -> braking { emergency },
    braking -> accelerating { end_emergency };
}
```




- Each discrete state has its own flow equations

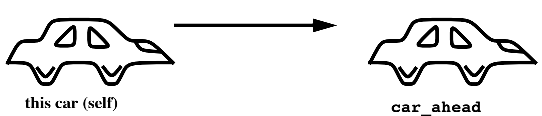




## Example: tracking law for follower

- The flow of a component can depend on other component’s outputs

```
type Car {
  input Car carAhead, ...
  output x;
  discrete
    following { v′ = f(x – x(carAhead) ) }
  ...
}
```








## Input, Output, State

- Numbers

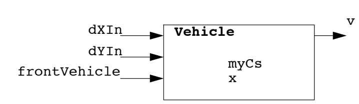
  *[continuous] number variable-name*
- Links to other components

  *type-name variable-name*
- Sets, Arrays

  *set(number) variable-name; array(number) variable-name;*

  *set(type-name) variable-name; array(type-name) variable-name*
- Examples

```
type Vehicle { ...
  input number dXIn, dYIn;
      Vehicle frontVehicle;
  state set(Control) myCs;
      number x;
  output number v;
... }
```



## Continuous Behavior

- Flow-equations

  *number-variable-name'= function-expression*

- Algebraic-definitions

  *number-variable-name = function-expression*

- Example

```
x' = w(a) + dxIn
z = 4 + x
```

- Note: We do not solve algebraic equations. The right-hand-sides of algebraic definitions cannot have circular dependencies





## Discrete Behavior

- Discrete states

  *state-name { equations }*

  - State specific flow equations and algebraic definitions

- Transitions - edges between states

  *from-state* **->** *to state*
  *{local-event, external-event}*
  when *{guard-expression}*
  do *{reset-expression}*

- Example

```
s1 -> s2 { merging,frontCar:mergeRequest} when { dXIn < 15 }
s1 -> s3 { mergeRequest } do { status := mergeEnabled }
```

- Terminology
  - executing a transition
  - synchronization
  - collective transition



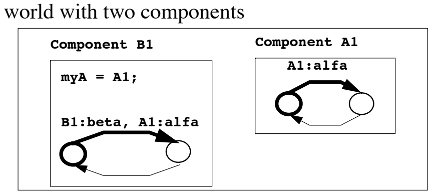


## Synchronization Rules

- A local-event must synchronize with all corresponding external-events
- Example

```
type A { ...
  discrete s1, s2;
  transition
      s1 -> s2 { alfa }                ... }
type B { ...
  state A myA;
  discrete s3, s4;
  transition
      s3 -> s4 { beta, myA:alfa }      ... }
```

- Consider a world with two components




- The two transitions must be synchronized



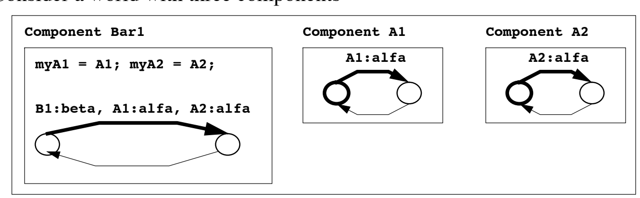


## More complicated example

- Example

```
type A { ...
  discrete s1, s2;
  transition
      s1 -> s2 { alfa }                            ...}
type B { ...
  state A myA1, myA2;
  discrete s3, s4;
  transition
      s3 -> s4 { beta, myA1:alfa, myA2:alfa}     ...}
```

- Consider a world with three components



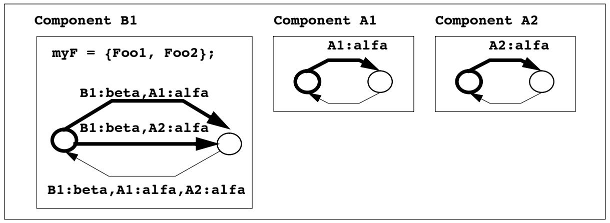

- All three transitions must be synchronized

## Synchronization Rules, ctd.

- External-event can be on a set. In that case it is possible to specify that **one** or **all** members must synchronize

```
type A { ...
  discrete s1, s2;
  transition   s1 -> s2 { alfa }                          ...}
type B { ...
  state set(A) myA;
  discrete s3, s4;
  transition   s3 -> s4 {beta, myA:alfa(one:t1)};
               s4 -> s3 {beta, myA:alfa(all)}             ...}
```

- Example








## Define Actions

- A set of variable assignments of the form:

  *type variable := expression;*

- Used to create local variables in transitions
- Define actions are executed sequentially
- Example

```
type A { ...
  output z := 1;
  discrete s1, s2;
  transition s1 -> s2 {alfa}
              define {number a := 2; number b := a + 1;}
              do { .. }
... }
```

- After execution “a = 2”, “b=3”
- Local variable scope is restricted to the action of a transition





## Do Actions

- A set of variable assignments of the form:

  *variable := expression*

- Order of evaluation
  - First all right hand sides are evaluated
  - Then values are assigned to left hand sides
- Example

```
type A { ...
  output z := 1;
  discrete s1, s2;
  transition s1 -> s2 {alfa}
        define   {number a := 2; number b := a + 1;}
        do       {z := b + 1;}
...}
type B { ...
  state A myA; number x;
  discrete s3, s4;
  transition s3 -> s4 {beta, myA:alfa}
        do       {x := z;}
...}
```

- After execution of collective transition z will be 4, x will be 1





## Setup -- Connect Actions

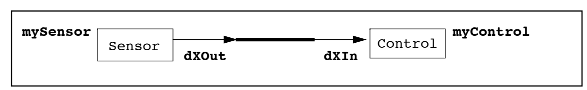
- Setup defines input-output and synchronization relationships

  *input-name(link-name) <- expression*

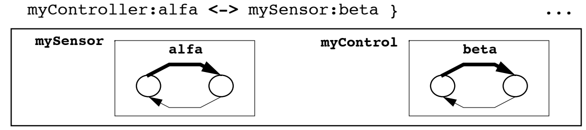
  *external-event <-> external-event*

- Example

```
type Vehicle { ...
  state  Sensor mySensor;Control myControl;
  setup  connect {
         deltaXIn(myController) <- deltaXOut(mySensor);
```




```
         myController:alfa <-> mySensor:beta }                ... }
```



## Setup, ctd

- Setup actions consist of *define*, *do* and *connect* actions
- Connections between events are dynamic. As the link variables change the event connections among components change
- Input-Output connections are partially static.
  - The left-hand-side, i.e. the input is set only once at creation time of the component
  - The right-hand-side, i.e. the output expression is evaluated dynamically





## Inheritance

- A subtype must honor all inputs, outputs, and exported events of the super-type
- It can add inputs, outputs, and events

  *type parent-name : child-name { ... }*

- Example

```
type A {
  input number x;
  output number y;
  export alfa;
  ...
}
type A : A{
  input number x, w;
  output number y, z;
  export alfa, beta;
  ...
}
```



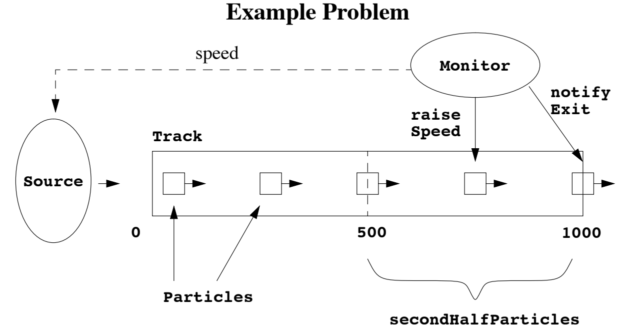


# SHIFT Example



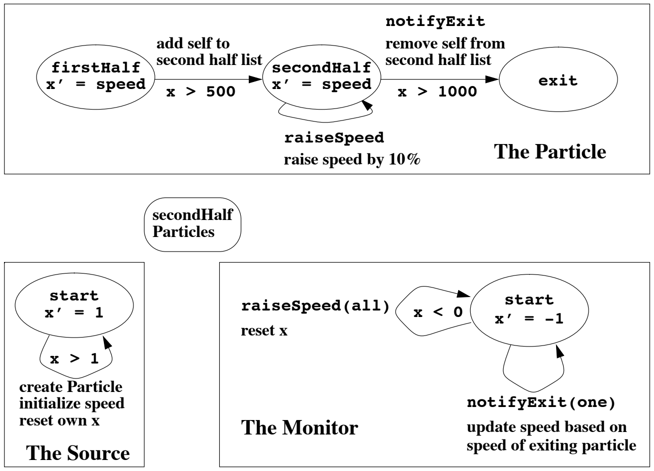


## Example Problem




- The source creates particles at regular intervals and sayings them an initial speed, as specified by the monitor. Particles move along a 1000m track. At random intervals the monitor commands the particles in the second half to accelerate by ten percent. Every time a particle exits the track, the monitor increases its specified speed as a function of the exiting particle's speed.

## The Design

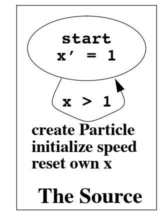







## The Implementation

```
#define nextBroadcastTime 5
global Monitor monitor := create(Monitor, speed := 100,
                x := nextBroadcastTime);
global Source source := create(Source, monitor := monitor);
global set(Particle) secondHalfParticles := {};

type Source
{
  state continuous number x;
        Monitor monitor;
  discrete start { x' = 1 };
  transition
    start -> start {}
      when x > 1
      do {
        create(Particle,
             speed := speed(monitor));
        x := 0;
      };
}
```

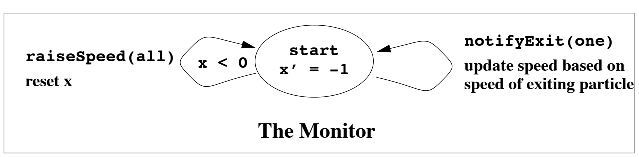







## The Implementation, ctd

```
type Monitor
{
  output number speed;
  state continuous number x;
  discrete start { x' = -1 }
  transition
    start -> start {secondHalfParticles:notifyExit(one:p)}
      do {speed := 0.5*(speed + speed(p));},
    start -> start {secondHalfParticles:raiseSpeed(all)}
      when x <= 0
      do {x := nextBroadcastTime;};
}
```

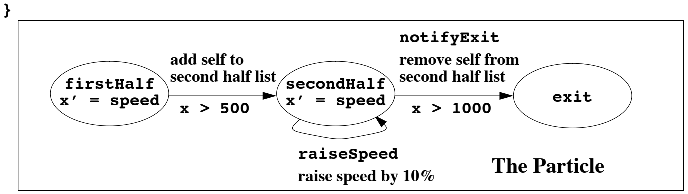







## The Implementation, ctd

```
type Particle {
  state continuous number x;
  output number speed;
  discrete firstHalf {x' = speed}, secondHalf {x' = speed}, exit;
  export raiseSpeed, notifyExit;
  transition
    firstHalf -> secondHalf {} when x >= 500
      do {secondHalfParticles := secondHalfParticles + {self}},
    secondHalf -> secondHalf {raiseSpeed}
      do {speed := 1.1*speed;},
    secondHalf -> exit {notifyExit} when x >= 1000
      do {secondHalfParticles := secondHalfParticles - {self};};
}
```



## The Synchronizations

```
type Monitor { ...
  transition
    start -> start {secondHalfParticles:notifyExit(one:p)}
      do {speed := 0.5*(speed + speed(p));};                ...}
type Particle { ...
  transition
    secondHalf -> exit {notifyExit}
      when x >= 1000
      do { secondHalfParticles := secondHalfParticles - {self};};...}


type Monitor { ...
  transition
    start -> start {secondHalfParticles:raiseSpeed(all)}
     when x >= duration
      do {x := nextBroadcastTime;};                          ...}
type Particle { ...
  transition
    secondHalf -> secondHalf {raiseSpeed}
      do { speed := 1.1*speed;};                             ...}
```



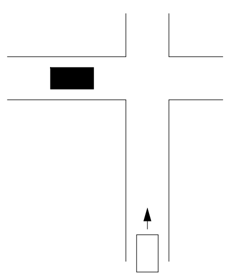


## SHIFT Example 2





## Colliding Vehicles

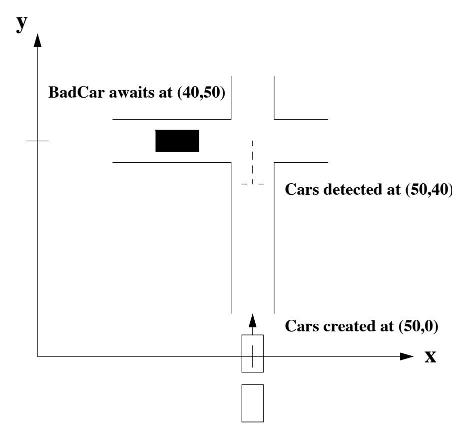


- Intersection of two one-way streets. White cars arrive at random intervals. When white cars are within 10m of the intersection the black car accelerates into the intersection and forces a collision.





## Design




- BadCar sets it speed to collide with the incoming Car

## The First Implementation

```
global set(Car) theCars := {create(Car, x:=50, y:=0, vy :=50, vx :=0)}
global BadCar theBadCar := create(BadCar)

type Car {
  input BadCar advesary;
  output continuous number x, y, vx, vy;
  flow default {x' = vx, y' = vy};
  discrete moving,
  export swerve;
  transition
    moving   -> crashed {swerve, advesary:now}
      do {vy       := 0;
          vx       := 0;
          theCars  := theCars - {self};          },
  }
```





## The First Implementation, ctd

```
type BadCar {
  output continuous number x:= 60, y := 50, vx, vy;
  state Car poorCar; continuous number t;
  flow default {x' = vx, y' = vy};
  discrete waiting, moving, crashed {t' = 1};
  export now;
  transition
  waiting -> moving {}
    when exists c in theCars : ( (y(c) > 40 ) and (vy(c) > 0) )
    do {vx           := -10 * ( vy(c)/(50-y(c)) );
        poorCar      := c;
        advesary(c)  := self;                              },
  moving -> crashed {now, poorCar:swerve}
    when ( (abs(y(poorCar) - y) < 1) and (abs(x(poorCar) - x) < 1 ))
    do {vx                 := 0;
        advesary(poorCar) := nil;
        poorCar           := nil;}
}
```





## The Second Implementation

```
global set(Car) theCars := {create(Car, x:=50, y:=0, vy :=50, vx :=0)}
global BadCar theBadCar := create(BadCar)

type Car {
  input BadCar advesary;
  output continuous number x, y, vx, vy;
  flow default {x' = vx, y' = vy};
  discrete moving,
          crashed  {vy' = -4, vx' = 3},  stoppedx {x' = 0, vy' = -4},
          stoppedy {y' = 0, vx' = 3},   stopped {x' = 0, y' = 0};
  export swerve;
  transition
    moving   -> crashed {swerve, advesary:now}
      do {vy       := vy/3;
          vx       := vx(advesary)/2 ;
          theCars  := theCars - {self};          },
    crashed  -> stoppedx {} when vx > 0 ,
    crashed  -> stoppedy {} when vy < 0 ,
    stoppedx -> stopped {} when vy < 0 ,
    stoppedy -> stopped {} when vx > 0 ;
}
```





## The Second Implementation, ctd

```
type BadCar {
  output continuous number x:= 60, y := 50, vx, vy;
  state Car poorCar; continuous number t;
  flow default {x' = vx, y' = vy};
  discrete waiting, moving, crashed {t' = 1};
  export now;
  transition
  waiting -> moving {}
    when exists c in theCars : ((y(c)>40) and (50>y(c)) and (vy(c)>0))
    do {vx           := -10 * ( vy(c)/(50-y(c)) );
        poorCar      := c;
        advesary(c)  := self;                              },
  moving -> crashed {now, poorCar:swerve}
    when ( (abs(y(poorCar) - y) < 1) and (abs(x(poorCar) - x) < 1 ))
    do {vx                 := 0;
        advesary(poorCar) := nil;
        poorCar           := nil;}
  crashed -> waiting {} when t > 0.5 do {t := 0; x := 60; y := 50;};
}
```

## The Third Implementation

```
function rn() -> number
global set(Car) theCars := {}
global Source s := create(Source);
global BadCar theBadCar := create(BadCar);

type Source {
  state continuous number t;
  discrete d {t' = -1};
  transition
    d -> d {} when t < 0
      define {number speed := rn();}
      do {t := 60/speed ;
          theCars := theCars +
                  {create(Car, x:= 50, y:= 0, vy := speed, vx :=0)};}
}
```





## SmartAHS





## Overview

- Summary of SmartAHS methodology
- Simulating a vehicle moving along the highway
    - Making use of the types provided by the SmartAHS libraries
        - Highway types
        - Vehicle type
        - Vehicle-Roadway-Environment Processor type
        - Controller type
        - AutomatedVehicle type
        - Sink and Source types
    - Creating Components
        - A two Section highway with a Source and Sink
        - Source creates vehicles as time pass





## The AHS Project

- The Automated Highway Systems project---to improve safety and reduce congestion---is funded in part by the U.S. Department of Transportation ($200M over 7 years)
- UC-Berkeley's California PATH is a partner in the nine-member AHS consortium along with General Motors, Bechtel, Parsons Brinckerhoff, Lockheed Martin, Delco, Hughes, Caltrans, and Carnegie Mellon University
- The consortium is responsible for designing, evaluating, and demonstrating a prototype AHS---with real cars on real roads
- In Europe---PROMETHEUS and DRIVE projects
- In Japan---RACS, AMTICS, VICS projects

## The PATH AHS Architecture-Five Layer Control Hierarchy

- Network Layer
  - End-to-end routing so vehicles reach their destinations without causing congestion---fluid flow and queuing models
- Link Layer
  - Highway segment control strategies to maximize throughput based on traffic state---activity flow models
- Coordination Layer
  - Communication protocols between vehicles and highway segments for coordinating vehicle maneuvers---finite state transition system models
- Regulation Layer
  - Observation subsystems and feedback controllers for safe execution of simple maneuvers such as join, split, lane change, entry, and exit--nonlinear control system models
- Physical Layer
  - Vehicle dynamical models---nonlinear differential equations





## The PATH AHS Architecture-Features

- Platooning
  - Separation within a platoon: 2m
  - Separation between platoons: 60m
- Simple coordinated maneuvers---join, split, lane change, entry, exit
- Network and link layers on highways
- Coordination, regulation, and physical layers on vehicles
- Fully automated, distributed, multi-agent control avoids single-point failures and provides flexibility
- Four-fold increase in transportation capacity along with enhanced safety





## Alternative Architectures

- The PATH design is one amongst many possibilities---Independent Vehicles, Cooperative Vehicles, Infrastructure Supported, Infrastructure Assisted, Maximally Adaptable
- A framework is needed in which these concepts can be designed and evaluated in a systematic manner





## Evaluation Tool Needs

- Several teams in different locations are developing and using these tools
- Tools must form a coherent suite---technical linkages between the tools must be clearly identified
- Developers and users must have a common understanding of the tools' interfaces and functionalities---technical linkages between the teams must be clearly identified
- Teams must be able to start their work as early as possible and continue their work as independently as possible---scheduling dependencies between different teams must be clearly identified

## Approach

- Standardized Tool Interface Formats (TIFs) between classes of tools
- Common understanding of the tools' interfaces and functionalities,
- Coherent tools
- Decoupled yet coordinated project plans.
- Tool developers develop tools to meet the interface formats
- Tool users organize their tool use in terms of interface formats
- SHIFT: programming language for specifying data and process models
- AHSTIF: AHS models specified in SHIFT
- SmartAHS: micro-simulation of these models using the SHIFT simulator





## Development Process: SmartAHS and Libraries

- What SmartAHS developers provide
    - The SHIFT simulation environment
    - The highway library that implements all highway types
    - The VREP type that maintains a vehicle's position on the highway
    - Simple examples of Sensor, Communication devices and their Environment Processors
    - Basic set of monitors that collect the necessary statistics to generate MOEs
- What library developers provide
    - Vehicle dynamics model library
    - Sensor models and detailed Environment Processor implementations
    - Actuator model library
    - Communication models and detailed Environment Processor implementations





## Development Process: Concept and Control Designers

- What concept designers will do
    - Can use vehicle, sensor, actuator, and communication device model libraries
    - Decide how to best represent a concept by a set of types
    - Specify the subsystem hierarchy and the input-output dependencies of automated highways and vehicles
- What control designers will do
    - Working within the guidelines set by concept designers, design and specify the control algorithms
    - Control designers can also provide a library of generic control models

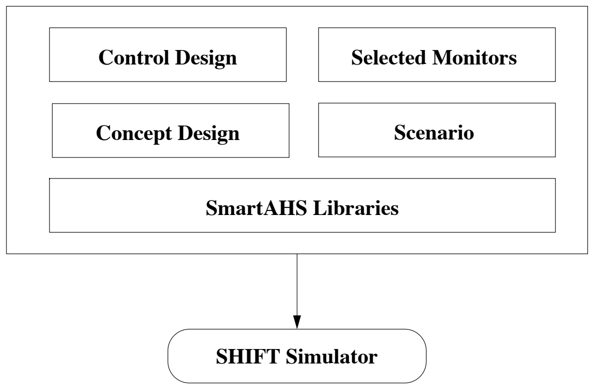




## Development Process: Scenario Specification

- What scenario specifiers will do
    - Implement detailed monitors to generate MOEs as necessary.
    - Using the highway library create highway representations
    - Using the traffic generator library create a traffic pattern
        - Final release of SmartAHS will have urban simulator package interfaces
    - Specify weather and road conditions
    - Select appropriate monitors to collect statistics for MOE generation

## How It All Comes Together








## Evaluation

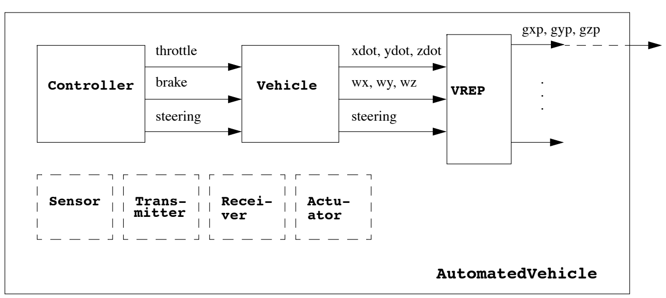
- Objective step
  - For each concept and for each scenario SmartAHS and other tools are used to generate a “column vector” of MOEs.
  - By varying the scenario for each concept a “matrix” of MOEs are obtained.
- Subjective step
  - The relevance of scenarios depend on location. Weights can be assigned to each scenario to create a “real-life” mix. This combines the different columns of the MOE matrix into one column vector.
  - The relevance of the MOEs depends on interest group. Weights can be assigned to each MOE. This combines the different rows of the MOE matrix into one row vector.
  - Combining the two steps assigns a “value” to the concept





## Example: Want to create Automated Vehicle

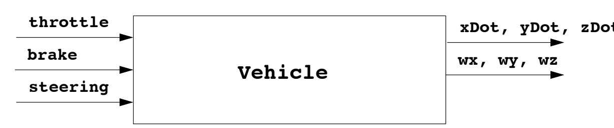
- Simple concept







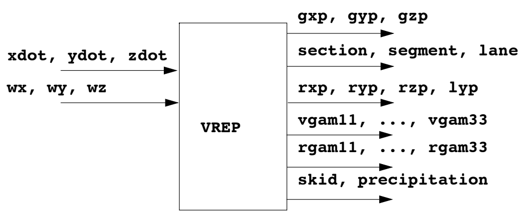

## The Vehicle Type




- Vehicle model has three inputs and six outputs
- wx, wy, wz are roll, pitch, yaw speeds of the vehicle in the vehicle coordinate frame
- Given the throttle, brake, and steering the vehicle computes
  - Its speed in the three directions in the “vehicle coordinate frame”
  - The speed with which the “vehicle coordinate frame” changes

## Vehicle Roadway Environment Processor

- This type maintains the position of a vehicle on the road and keeps the relationships between the three coordinate frames
- Three coordinate frames: vehicle, roadway, global








## Sample VREP Fragments

- Declare data model of the VREP

```
type VREP
// Vehicle Roadway Environment Processor
{
  input continuous number xDot, yDot, zDot;
      // vehicle’s longitudinal, lateral and longitudinal speeds
      continuous number wx, wy, wz;
      // vehicle’s roll, pitch and yaw speeds
```

- Sample flow equations that maintain the relations of vehicle and world coordinate frames

```
/* $mat2scalar
                                  |vgam11 vgam12 vgam13|
|gxp gyp gzp|’ = |xDot yDot zDot|*|vgam21 vgam22 vgam23|
                                  |vgam31 vgam32 vgam33|;

|vgam11 vgam12 vgam13|’    | 0  wz -wy| |vgam11 vgam12 vgam13|
|vgam21 vgam22 vgam23| =   |-wz  0  wx|*|vgam21 vgam22 vgam23|
|vgam31 vgam32 vgam33|     | wy -wx 0 | |vgam31 vgam32 vgam33|;
```





## VREP Code Continued

- Code fragment that maintains the “Section” a vehicle is in

```
transition
    cruise -> cruise{updateSection}
    when
        rxp > length(section) and ...
    define {
        ld := laneDown(lane)[followLane];
        secd := section(ld);
        segd := segments(secd)[0];
        gs := grade(segd);
        bs := banking(segd); }
    do {
        rxp := rxp - length(section);
        ryp := ryp - previousYOffset(secd);
        skid := skid(segd);
        ...
        lane := ld;
        section := secd;
        segment := segd; };
```



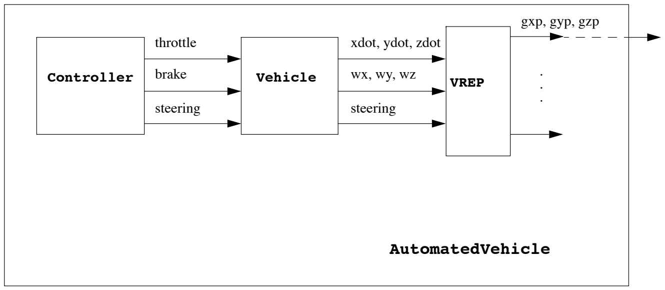


## Controller

- To be implemented by concept designers
  - Generates throttle, steering, brake
  - Requires concept development
- We have selected two cases:
  - throttle := 10, steering := 0, brake := 0
  - throttle := 10, steering := 0.005, brake := 0
- Control designers have to implement complete solutions

## Automated Vehicle

- Uses off-the-shelf components








## AutomatedVehicle Code

- Declare data model

```
type AutomatedVehicle
{
  output Source source;
      Sink sink;
      Vehicle vehicle;
      VREP vrep;
      Controller controller;

      continuous number gxp, gyp, gzp;
      number width := 2.5, length := 5.0;

  export acceptExitingVehicle;

  discrete travel, exit;
```





## AutomatedVehicle, ctd.

- Initialize subsystems

```
setup do
    {
        vehicle := create(Vehicle,
                xDot := 23.0,
                I_e := 0.15);
        vrep := create(VREP,
                gxp := gxp(source), gyp := gyp(source),gzp := gzp(source),
                section := section(source),segment := segment(source),
                lane := lane(source),
                rxp := rxp(source),ryp := ryp(source),rzp := rzp(source),
                lyp := lyp(source),
                vgam11 := vgam11(source),
                ...
                vgam33 := vgam33(source),
                vehicle := self);
        controller := create(Controller);
    }
```





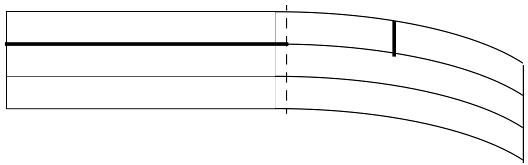
## AutomatedVehicle, ctd.

- Interconnect subsystems

```
do      {
        AutomatedVehicles := AutomatedVehicles + {self};
        }
connect {
        xDot(vrep)          <- xDot(vehicle);
        yDot(vrep)          <- yDot(vehicle);
        zDot(vrep)          <- zDot(vehicle);
        wx(vrep)            <- wx(vehicle);
        wy(vrep)            <- wy(vehicle);
        wz(vrep)            <- wz(vehicle);

        followLane(vrep)    <- followLane(controller);

        throttle(vehicle)   <- throttle(controller);
        steering(vehicle)   <- steering(controller);
        brake(vehicle)      <- brake(controller);
    };
```

## Creating The Highway

- Highway Types
  - Sections consist of Lanes and Segments
  - Segments define the geometry
  - There may be Barriers and Blocks



- Highway types have a “road reference frame”, they reside in the “global reference frame”
- Example highway layouts exist!





## Sink and Source

- Source and Sink must be placed on the highway
- Source creates “AutomatedVehicles” that travel to the Sink





## Creating The Simulation

- Must create a highway.
  - Instantiates a HighwayBuilder component that creates a 2 section highway
- Must create a source and a sink
  - Instantiates one component of each and places them on the highway
- The simulator takes over!





## How Do We Generalize

- “EnvironmentProcessor”s for Sensor, Receiver types exist.
- Detailed Sensor, Transmitter, Receiver models to be provided by various B5 team members.
- Control specifications and AutomatedVehicles and AutomatedHighways to be specified by Concept Developers.
- Scenario specifiers to develop highway layouts, traffic patterns and types that collect statistics for MOEs.
- C2, C3, C4 and evaluation team to combine them in various simulation runs.

# The Mathematical Model

... a set of slides are still missing here ...





# Type in the Math Model

```
type H1 {
  discrete   q1 {flow1},
             q2 {flow2},
             q3 {flow3};
  state continuous number x1, x2;
  state      H1 C0_1;
             H2 C0_2;
             set(H1) C1;
             set(H3) C2;
  export     l1, l2, l5;
  flow  flow1 {x1' = 3, x2' = x1}
        flow2 {x1' = x2(C0_1)}
        flow3 {x2 = 3 + x2(C0_1)}
  transition
        q3 -> q2 {l1}
        q3 -> q1 {C0_1:l2}
        q2 -> q1 {C2:l3(all)}

        q1 -> q2 {C1:l5(one)}

        q1 -> q2 {} when x2 >= 25
```

```
type Car {
  discrete   accel {flow1},
             decel {flow2},
             cruise {default};
  state continuous number x, v;
  state  Car fCar;
         Wheel myWheel;
         set(Car) neighbors;
         set(People) passengers;
  export slow, fast, brake;
  flow   flow1 {v' = 3}
         flow2 {v' = -2}
         default {v' = 0; x' = v}
  transition
    cruise -> decel {slow}
    cruise -> accel {fCar:fast}
    decel -> accel
      {passengers:wakeup(all)}
    accel -> decel
      {neighbors:brake(one)}
    accel -> decel when v >= 25
```





# Instantiation of a Component

```
type H1 {
  discrete   q1{flow1}, q2{flow2}
             q3 {flow3};
  state continuous number x1, x2;
  state      H1 C0_1;
             H2 C0_2;
             set(H1) C1;
             set(H3) C2;
  export     l1, l2, l5;
  flow  flow1 {x1' = 3, x2' = x1}
        flow2 {x1' = x2(C0_1)}
        flow3 {x2 = 3 + x2(C0_1)}
  transition
        q3 -> q2 {l1}
        q3 -> q1 {C0_1:l2}
        q2 -> q1 {C2:l3(all)}
        q1 -> q2 {C1:l5(one)}

        q1 -> q2 {} when x2 >= 25
        q2 -> q1 {} when exist...
```

**component (H1 1) {**
  **discrete =** q1

  x1 = 3; v = 25;
  $C_{0\_1}$ = (H1 2);
  $C_{0\_2}$ = (H2 1);
  $C_1$ = { (H1 2) (H1 3) }
  $C_2$ = { (H3 1) (H3 2) }
  export l1, l2, l5;
  flow    flow1 {x1' =3; x2' = 3}

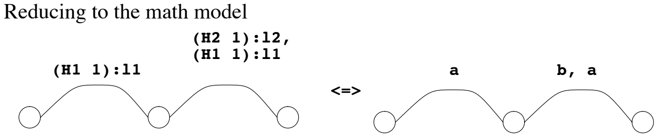
  transition
    q1->q2 {(H1 1):l1}
    q3->q1 {(H1 2):l2}
    q2->q1 {(H3 1):l3, (H3 2):l2}
    q1->q2 {(H1 2):l5}
    q1->q2 {(H1 3):l5}
    q1->q2 {} when v >= 25
    q2->q1 {} ..
    q2->q1 {} ..



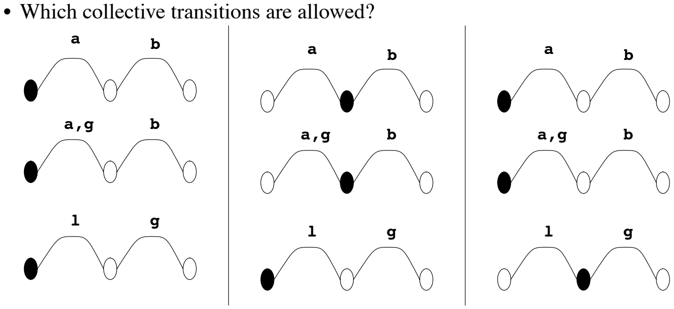


# Synchronization Example

- Reducing to the math model




- Which collective transitions are allowed?



## What's Available

- http://www.path.berkeley.edu/shift
- The shic compiler which takes a SHIFT specification and produces a C file that can be compiled and linked with other C files
- A command line (tty) debugger that allows you to inspect a SHIFT program's entities at run time
- A Tcl/Tk based graphic environment that makes it easier to run and visualize simulations and debug SHIFT programs by graphically inspecting various entities at run time.
- A C Application Program Interface
- Documents
- Libraries of components for Automated Highway Systems





## Current Applications

- Detailed specifications and design of Vehicle / Highway control systems
- Design and simulation of formation of mini-submarines for marine survey
- Design and simulation of air traffic control system
- Xerox machine control modeling





## Planned Enhancements to SHIFT

- Better, faster integration algorithms
-