An Object-Oriented Modeling Language for Hybrid Systems

Huixing Fang Huibiao Zhu Jianqi Shi

East China Normal University
fang.huixing@gmail.com

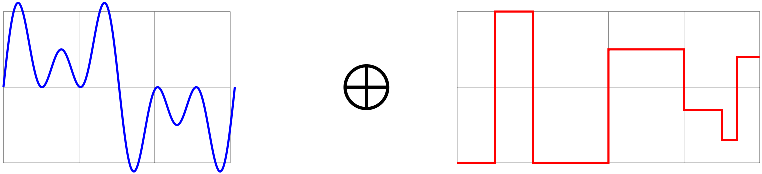
Outline

1. Introduction on Hybrid Systems
2. Motivation of Our Work
3. Object-Oriented Modeling Language
4. Conclusion and Future Work

Outline

1. Introduction on Hybrid Systems
2. Motivation of Our Work
3. Object-Oriented Modeling Language
4. Conclusion and Future Work

Hybrid Systems



Hybrid systems consist of discrete control programs and continuous physical behaviour, such systems exhibit both continuous and discrete dynamic behaviour.

Hybrid Automata

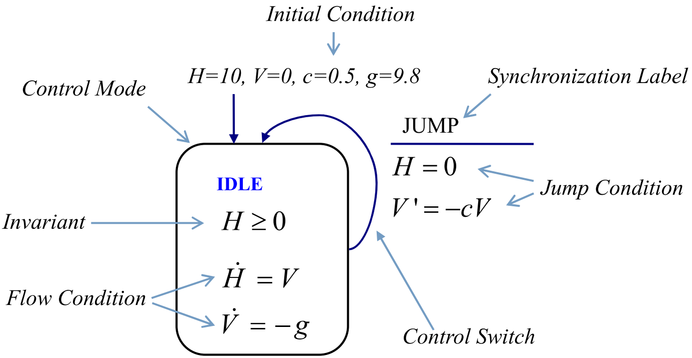


Figure : Hybrid automaton of a bouncing ball

Outline

1. Introduction on Hybrid Systems
2. Motivation of Our Work
3. Object-Oriented Modeling Language
4. Conclusion and Future Work

Motivation

Automata-Based Languages/Notations

- not the best choice for creating models from building blocks;
- the differential equations, invariants and difference equations are tightly coupled.

Motivation

Automata-Based Languages/Notations

- not the best choice for creating models from building blocks;
- the differential equations, invariants and difference equations are tightly coupled.

Process-Algebra-Based Languages/Notations

- have solid theoretical foundation of formal analysis
- not accepted widely by designers and developers due to the complicated symbols, mathematical abstractions and various concept abbreviations.

Motivation

Automata-Based Languages/Notations

- not the best choice for creating models from building blocks;
- the differential equations, invariants and difference equations are tightly coupled.

Process-Algebra-Based Languages/Notations

- have solid theoretical foundation of formal analysis
- not accepted widely by designers and developers due to the complicated symbols, mathematical abstractions and various concept abbreviations.

State-Chart-Based Languages/Notations

- de facto approach for model-based development of embedded systems
- weak on model reuse as well, e.g., the continuous-time chart in Simulink/Stateflow cannot be reused

Outline

1. Introduction on Hybrid Systems
2. Motivation of Our Work
3. Object-Oriented Modeling Language
4. Conclusion and Future Work

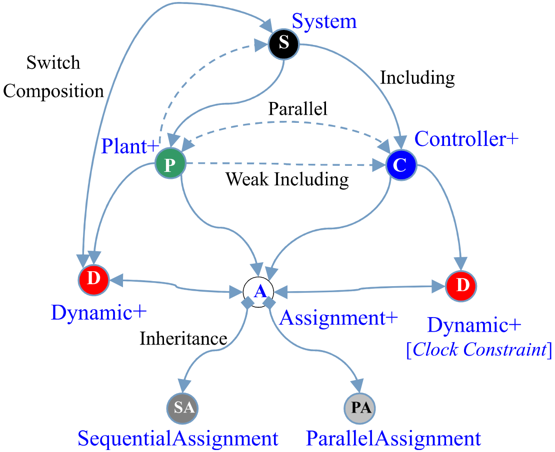
Apricot

Syntax

System	::=	$(\|\|_{i=1}^{n}$ Plant$_i$) $\|$ ($\|\|_{i=1}^{m}$ Controller$_i$);
Plant	::=	AtomComp | (AtomComp $\|$ Controller) | HierComp
		| (HierComp $\|$ Controller);
Controller	::=	AtomComp;
AtomComp	::=	*Comp*(Condition$^+$, Dynamic$^+$, Assignment$^+$);
HierComp	::=	*Comp*(Condition$^+$, Dynamic$^+$, Assignment$^+$, System);
Assignment	::=	SequentialAssignment | ParallelAssignment.

- *Dynamic*$^+$ represents a set of *Dynamic* objects used for continuous dynamics;
- *Assignment*$^+$ for *Assignment* objects denoting discrete behaviors;
- $||$ denotes parallel composition. $||_{i=1}^{n}$ *Plant*$_i$ represents the parallel composition of n plants;
- *Comp*$(\cdot)$ represents the control switch composition relation under the condition in *Condition*$^+$.

The relationship of objects in Apricot



→	Including
- - →	Weak Including
↔	Control Mode Switch Composition
← - →	Parallel Composition
◆→	Inheritance
S	System
P	Plant
C	Controller
D	Dynamic
A	Assignment

the weak include relationship, i.e., the object at starting point may not contain an object at the ending point, e.g., a plant may not contain a subsystem.

Continuous Behavior

Differential Equations:

```
void Continuous(){
    dot(Var_1, Nat_1) == MathExp_1 ;
                  ...
    dot(Var_n, Nat_n) == MathExp_n ;
}
```

where, Var_i is a continuous variable, Nat_i represents the derivative order of Var_i, $MathExp ::= Function(Vars, \dot{V}ars)$, a mathematical function, e.g., sine, cosine, log.

Discrete Behavior

Discrete Assignments:

```
void Discrete(){
    Variable_1 = MathExp_1 ;
        ...
    Variable_n = MathExp_n ;
}
```

- ParallelAssignment: the parallel composition of the assignment statements;
- SequentialAssignment: sequential composition of assignment statements. if-statement, for-loop, and method-call, etc.

Interfaces-System

```
interface System{
    Requires plants[1..*] : Plant;
    Requires controllers[1..*] : Controller;
    void Init();
}
```

- *Requires* is a keyword for the declaration of variable-requirement,
- $1..*$ denotes the amount of entities $>= 1$.
- method *Init*() indicates that the System has an initializer without any argument nor value return.

Interfaces-Plant

```
interface Plant{
   Requires dy[1..∗] : Dynamic;
   Requires ass[1..∗] : Assignment;
   Requires sy[0..1] : System;
   Requires controller[0..1] : Controller;
   void Composition(){
     Requires coms[1..∗](?|!)[0..1] : (dss[1..∗] : Dynamic|System,
                 ass[0..1] : Assignment, dsd[1..∗] : Dynamic|System)
                 { Condition{}; Discrete{}; };
   };}
```

- ! and ? behind the composition name denote the asynchronous communication between different components. ! used for asynchronous message sending;
- It is the case of synchronous communication when ? and ! are absent;
- We can define the synchronization of two compositions A and B explicitly, as $A \mid\mid B$; and $A \sim B$ denotes asynchronous communication between A and B.

Interfaces-Controller

```
interface Controller{
    Constraint clock;
    Requires dy[1..*] : Dynamic;
    Requires ass[1..*] : Assignment;
    void Composition(){
      Requires coms[1..*](?|!)[0..1] : (dys[1..*] : Dynamic,
               ass[0..1] : Assignment, dyd[1..*] : Dynamic)
               { Condition{}; Discrete{}; };
    }; }
```

- the constraint-indication *Constraint clock* denotes that: the derivative order assigned to the variables in the Dynamic object of Controller must be a constant number 1, the derivative is also a constant number.

Polymorphism

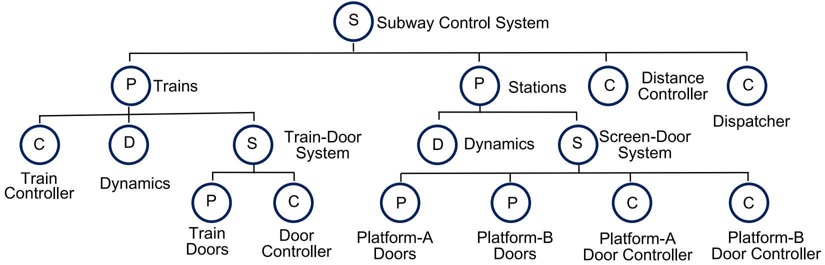


Figure : A hierarchical structure of the subway control system. The capital letters ‘S’, ‘P’, ‘C’, ‘D’ denote the entities of system, plant, controller, and dynamic, respectively

Polymorphism-Example

```
class ScreenDoorSystem extends DoorSystem{
  ...
  void Init(){
    registerComposition(PADoors,PADController);
    registerComposition(PBDoors,PBDController);
    PADoors.closed.Start();
    PBDoors.closed.Start();
    PADController.closed.Start();
    PBDController.closed.Start();
  }
}
```

```
class TrainDoorSystem extends DoorSystem{
  ...
  void Init(){
       registerComposition(doors,TDController);
       //start dynamic closed
       doors.closed.Start();
       //start controller for train doors
       TDController.closed.Start();
  }
}
```

ScreenDoorSystem and *TrainDoorSystem* have the same type of *DoorSystem*, but they can behave differently when *Init* is called.

Code-Reuse

Consider the Train Automaton:

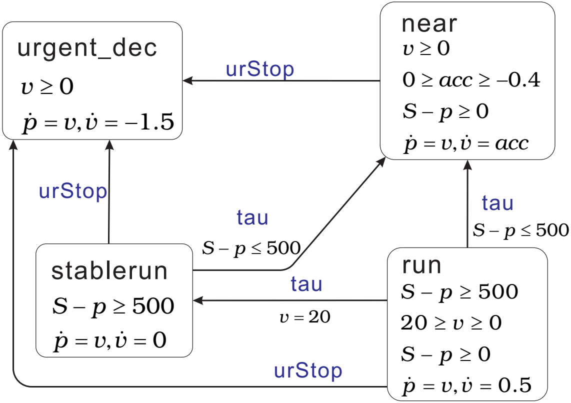


Figure : Some parts of the hybrid automaton. Variable S denotes the position of the station in front of the train. The position of the train is represented by variable p, the velocity is by v. Since $0 \geq acc \geq -0.4$ (m/s^2), variable *acc* denotes the deceleration of the train in the mode *near*

Code-Reuse

The declaration of continuous behavior of the train in Apricot.

```
class TrainBehavior implements Dynamic{
 Real p;//position
 Real v;//velocity
 Interval acc;//acceleration
 real pos1; //The lower bound of the position
 real pos2; //The upper bound of the position
 real vel1; //The lower bound of the velocity
 real vel2; //The upper bound of the velocity
  TrainBehavior(Real p, Real v, Interval acc,
   real pos1, real pos2, real vel1, real vel2){
   this.p = p; this.v = v; this.acc = acc;
   this.pos1 = pos1; this.pos2 = pos2;
   this.vel1 = vel1; this.vel2 = vel2;
  }
...//Some extra codes
 void Continuous(){
  //Declares the continuous behavior
     dot(p,1) == v;
     dot(v,1) == acc;
  }
 Invariant{
    p in [pos1, pos2];
    v in [vel1, vel2];
  };
}
```

Code-Reuse

For Mode Run

the mode *run* in Train Automaton can be declared as:

`run = new TrainBehavior(p, v, [0.5,0.5], -Inf, S-500, 0, 20).`

For Mode Near

the mode *near* can be instantiated in our language as:

`near = new TrainBehavior(p,v,[-0.4,0],-Inf,S,0,Inf).`

For Control Switches from *run* and *stablerun* to *near*:

```
tau({run,stablerun},,near){
 Condition{
    abs(S-p) <= 500;
  };
}
```

Complex in Hybrid Automaton:

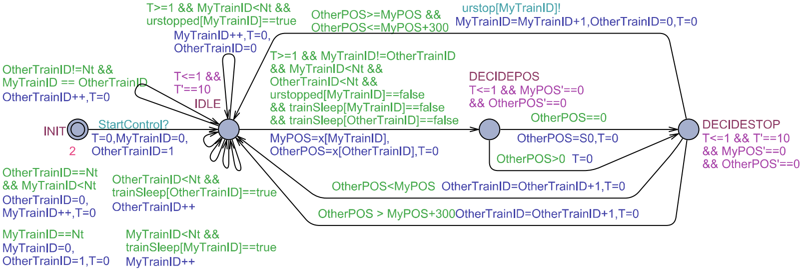


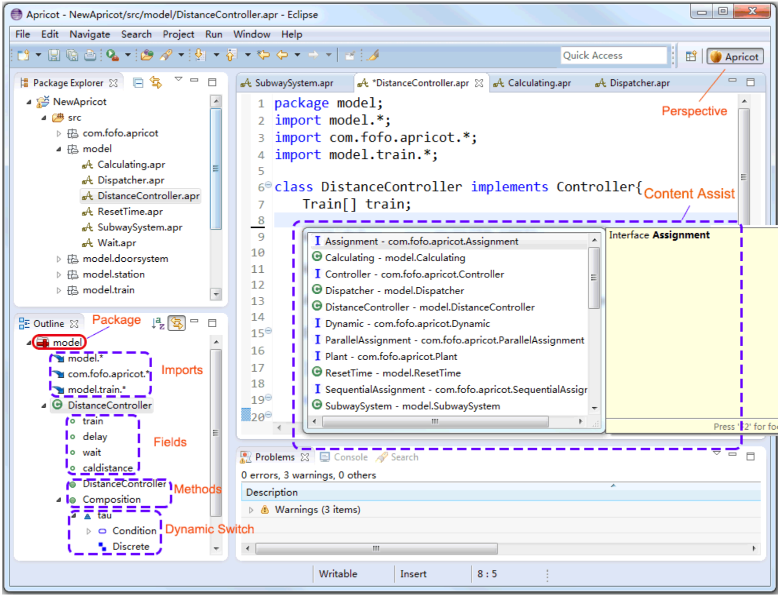
Figure : The hybrid automaton for urgent control of subway trains in UPPAAL.

Better Declaration for Control Logic

Better in Apricot with traditional control structures (for-loop, if):

```
package model;
import com.fofo.apricot.SequentialAssignment;
import model.train.Train;
class Calculating implements SequentialAssignment{
 Train[] train;  //the array of trains
 Calculating(Train[] train){ //constructed by an array of trains dynamically
   this.train = train;
 }
 void Discrete(){
   int mindis = Inf;
   for(Train currTrain : train){
     int currdir = currTrain.getCurrentDirection();
     real currpos = currTrain.getCurrentPosition();
     for(Train otherTrain : train){
       int otherdir = otherTrain.getCurrentDirection();
       real otherpos = otherTrain.getCurrentPosition();
       if(currTrain!=otherTrain and currdir == otherdir){
         //when two trains are different, but in same direction
         //calculate the distance between them
         real distance = currdir * (otherpos - currpos);
         if(distance<=300 and distance>=0 and distance < mindis){
           mindis = distance;
         }
       }//end calculate of distance
     }
     if(mindis < Inf){//the initial value of mindis is Inf
       currTrain.urStop(!);//stop currTrain
       mindis = Inf;
     }
     else if(currTrain in currTrain.urgent_stop){//currTrain can be restart
       currTrain.urStart(!);
     }
   }//end for-loop
 }//end discrete()
}//end class Calculating
```

Modeling Tool



Outline

1. Introduction on Hybrid Systems
2. Motivation of Our Work
3. Object-Oriented Modeling Language
4. Conclusion and Future Work

Conclusion and Future Work

Provide

- Object-Oriented Language for Hybrid Systems
- Code-Reuse, Better Control Logic Implementation

Future Work

- Developing Formal Verification Tool for Apricot
- Simulation and Animation with Apricot

Thanks

Thank you very much!