Apricot: An Object-Oriented Modeling Language for Hybrid Systems

Huixing Fang[1], Huibiao Zhu[1], and Jianqi Shi[2]

[1] Shanghai Key Laboratory of Trustworthy Computing
Software Engineering Institute, East China Normal University, China
[2] School of Computing, National University of Singapore, Singapore
{wxfang,hbzhu}@sei.ecnu.edu.cn, shijq@comp.nus.edu.sg

Abstract. Hybrid systems arise in embedded control from the interaction of continuous physical behavior and discrete digital controllers. In this paper, we propose *Apricot* as an object-oriented language for modeling hybrid systems. The language combines the features of domain-specific and object-oriented languages, which fills the gap between design and implementation. With respect to the application of *Apricot*, we propose the model for urgent distance control in subway control systems. In addition, the comparison with hybrid automata is discussed, which indicates the scalability and conciseness of the *Apricot* model. Moreover, we develop a prototype modeling tool (as a plug-in for Eclipse) for our language. According to the characteristics of object-orientation and the component architecture of *Apricot*, we conclude that it is suitable for modeling hybrid systems without losing scalability.

1 Introduction

Hybrid systems consist of discrete control programs and continuous physical behaviour, such systems combine logical decision making with the generation of continuous-valued control laws. Usually, hybrid systems modeled as finite state machines are coupled with partial or ordinary differential equations, and difference equations. In order to model hybrid systems, numerous modeling approaches have been proposed: hybrid automata [1], Hybrid CSP [10,18], and hybrid programs [12]. With respect to the formal verification of hybrid systems, various tools can be used, e.g., HyTech [11], d/dt [3], PHAVer [7], ProHVer [17], SpaceEx [8], and KeYmaera [13]. The common feature of these works is that most of them focus on the high-level abstraction of hybrid systems. However, in industry, both abstraction and easy usability are important. Therefore, there is a demand for developing a modeling language that caters for these two concerns for hybrid systems. Nowadays, automata, process algebras and statecharts are the three main notations for modeling hybrid systems.

First, automata-based notations are not suitable for creating models from building blocks. The main reason is that, in automata the differential equations, invariants and difference equations are tightly coupled. This kind of design style is good for small-scale models since the designer does not need to reuse declarations between different system components in this scenario. However, in industrial models, there are thousands of trains, cars, digital controllers, and wireless sensors. Thus, for realistic applications, we have to consider how to reuse and extend models, components, control laws.

Secondly, process-algebra-based approaches are suitable for hybrid systems as the theoretical foundation of formal analysis. However, in industry, they are not accepted widely by designers and developers. Because, the notation usually contains complex symbols, mathematical abstractions and various concept abbreviations. Expert help and guidance is often required by industrial practitioners in order to successively apply such formalisms.

Thirdly, in industry, statechart-based notation is a de facto approach for model-based development of embedded systems. It is similar to automata-based notations, but equipped with high-level programming language features. With plenty of tool support for simulation, testing and code generation, the model-based development underpinning for hybrid systems can be implemented efficiently. However, the statechart approach is also weak on model reuse. For instance, the continuous-time chart in Matlab Simulink/Stateflow cannot be reused. In addition, it is also difficult to support inheritance of components in this kind of notations.

Apricot is a modeling language that has a clear and simple syntax, an appropriate architecture for constructing models for hybrid systems, and an explicit semantics. The contributions of our work can be elaborated as follows.

(i) Innovation on the *interface* concept with respect to interfaces of traditional object-oriented languages: variable-requirements, constraint-indications and built-in block statements are allowed in interface declaration. The variable-requirements define the relationships between different types. Therefore, there have the ability to describe the ownership among different objects. The constraint-indications denote the assumptions that the behavior of the system is forced to conform. The built-in block statements denote the right usage and position that the statement should be in. As a consequence, our innovation here enhances and clarifies the relationship of various components by variable-requirements, specifies the limitation of components by constraint-indications, and explicitly describes the proper usages of blocks by the built-in block statement declarations.

(ii) We apply the principle of *Architecture as Language*, which combines the features from Domain-Specific (DSL [16,6]) and Object-Oriented Languages (OOL). The DSL notations employed in *Apricot* are appropriate for building component architecture. As a result, it makes it easier to interact with domain experts during the design process. On the other hand, OOL is familiar to developers in industry. The combination of DSL and OOL in *Apricot* fills the gap between the design at the abstract level and the implementation at the concrete level.

(iii) For tool support, we implement the *Apricot* language with Xtext[1] on Eclipse, and develop a prototype tool *XtextApricot*[2] for modeling hybrid systems.

This paper is organized as follows. Section 2 is an overview of hybrid systems. In Section 3, we propose the syntax of *Apricot*. Section 4 presents our case study : a subway control system. The operational semantics is described in Section 5. Related work is discussed in Section 6. Finally, we draw our conclusions in Section 7.

2 Background of Hybrid Systems

Hybrid systems consist of discrete control programs and continuous physical behavior, such systems exhibit both continuous and discrete dynamic behavior. We illustrate the concept of hybrid systems by the definition of hybrid automata as follows.

Definition 1 (Hybrid Automata). *A hybrid automaton is a tuple* $HA = \langle V, M, F, I, \eta, E, J, \Sigma, \sigma \rangle$*, where:*

- $V = \{v_1, v_2, \cdots, v_n\}$ *is a finite set of* n *real-valued variables, where* n *is the dimension of* HA*.*
- *For each control mode* $m \in M$*, flow condition* $F(m)$ *is a predicate over the variables in* $V \cup \dot{V}$*, where* $\dot{V} = \{\dot{v_1}, \dot{v_2}, \cdots, \dot{v_n}\}$ *and* $\dot{v_i}(1 \leq i \leq n)$ *is the first-order derivative of* v_i *with respect to time.*
- *For* $m \in M$*, the invariant condition* $I(m)$ *for mode* m *is a predicate over the variables in* V*.*
- *For* $m \in M$*, the initial condition* $\eta(m)$ *is a predicate over the variables in* V*.* $\eta(m)$ *sets the initial state of* m*.*
- *For* $m, m' \in M$*, if* HA *has a transition from* m *to* m'*, then* $(m, m') \in E$ *is a control switch.*
- *For each control switch* $e \in E$*, the jump condition* $J(e)$ *is a predicate over the variables in* $V \cup V'$*. For* $1 \leq i \leq n$*,* $v_i' \in V'$ *refers to the new value of the variable* v_i *as soon as the control switch had finished.*
- *Events.* Σ *is a finite set of events.*
- *Synchronization labels. For each control switch* $e \in E$*, the synchronization label* $\sigma(e)$ *is an event in* Σ*.*

State of Hybrid Automata: If m is a control mode, $r = (r_1, \cdots, r_n) \in \mathbb{R}^n$ is a value of variables in V, then the pair (m, r) is a state of the hybrid automaton HA. The state (m, r) is valid only if $I(m)$ is true when the value of the variable v_i is r_i, for $1 \leq i \leq n$.

Parallel Composition of Hybrid Automata: Given hybrid automata:

$$HA_1 = \langle V_1, M_1, F_1, I_1, \eta_1, E_1, J_1, \Sigma_1, \sigma_1 \rangle,$$

and

$$HA_2 = \langle V_2, M_2, F_2, I_2, \eta_2, E_2, J_2, \Sigma_2, \sigma_2 \rangle.$$

1 An open-source framework: `http://www.eclipse.org/Xtext/`

2 Available at `http://www.apricotresearch.com/`

Let $HA_{1||2} = \langle V, M, F, I, \eta, E, J, \Sigma, \sigma \rangle$ be the parallel composition of HA_1 and HA_2. Thus, $M = M_1 \times M_2$, $V = V_1 \cup V_2$, $\sigma = \sigma_1 \cup \sigma_2$. The most important point is that, if $\exists e \in E, \sigma(e) \in \Sigma$, $e_1 \in E_1$ and $e_2 \in E_2$ associate with the same synchronization label $\sigma(e)$, then e_1 and e_2 should be synchronized in $HA_{1||2}$.

3 Syntax of Apricot

As a modeling language for hybrid systems, it is required to consider the hierarchical structures of a system to demonstrate the modularity features. Also, we need to propose the definitions of system dynamics with the relations between continuous flows and discrete assignments.

3.1 Architecture and Fundamental Syntax

Here, we will give an overview of our language. The following recursive definitions cover the overview architecture of *Apricot*.

$$
\begin{aligned}
\texttt{System} &::= (\|_{i=1}^{n} \texttt{Plant}_i) \,\|\, (\|_{i=1}^{m} \texttt{Controller}_i);\\
\texttt{Plant} &::= \text{AtomComp} \;|\; (\text{AtomComp} \;\|\; \texttt{Controller}) \;|\\
&\quad\;\; \text{HierComp} \;|\; (\text{HierComp} \;\|\; \texttt{Controller});\\
\texttt{Controller} &::= \text{AtomComp};\\
\text{AtomComp} &::= \mathit{Comp}(\texttt{Condition}^{+}, \texttt{Dynamic}^{+}, \texttt{Assignment}^{+});\\
\text{HierComp} &::= \mathit{Comp}(\texttt{Condition}^{+}, \texttt{Dynamic}^{+}, \texttt{Assignment}^{+}, \texttt{System});\\
\texttt{Assignment} &::= \texttt{SequentialAssignment} \;|\; \texttt{ParallelAssignment}.
\end{aligned}
$$

where $n, m \in \mathbb{Z}^+$ (positive integers), '$\|$' denotes parallel composition. '$\|_{i=1}^{n}$ $Plant_i$' represents the parallel composition of n plants. '$Comp(\cdot)$' represents the control mode (i.e., *Dynamic* in *Apricot*) switch composition under the condition in '$Condition^{+}$'. In this paper, the control mode switch composition relationship is abbreviated as *composition relationship*. It declares the relation between continuous flow and discrete assignment. Continuous flow and discrete assignment are declared by the classes implementing interfaces *Dynamic* and *Assignment*, respectively. Here, '$Dynamic^{+}$' represents a set of *Dynamic* objects, and '$Assignment^{+}$' is for *Assignment* objects. The discrete assignment for continuous variables during the control switch, usually, is used as an abstraction for currently undetermined physical behavior.

In Fig. 1, the relationship of objects in *Apricot* is illustrated as a relation graph. The arrow '$\rightarrow$' represents the include relationship from starting point to the ending point. '$\dashrightarrow$' is the weak include relationship, i.e., the object at starting point may not contain an object at the ending point. The '$\leftrightarrow$' represents the composition relationship (i.e., control mode switch composition). '$\dashleftarrow\!\dashrightarrow$' is for the parallel composition relationship. The inheritance relationship is denoted by '$\diamond\!\!\rightarrow$'. Both *SequentialAssignment* and *ParallelAssignment* inherit interface *Assignment*. Each system contains one or more plants and controllers. *Dynamic* object is an instance of the class that implements the *Dynamic* interface, referring to continuous flow which is used to model continuous behaviors of physical plants.

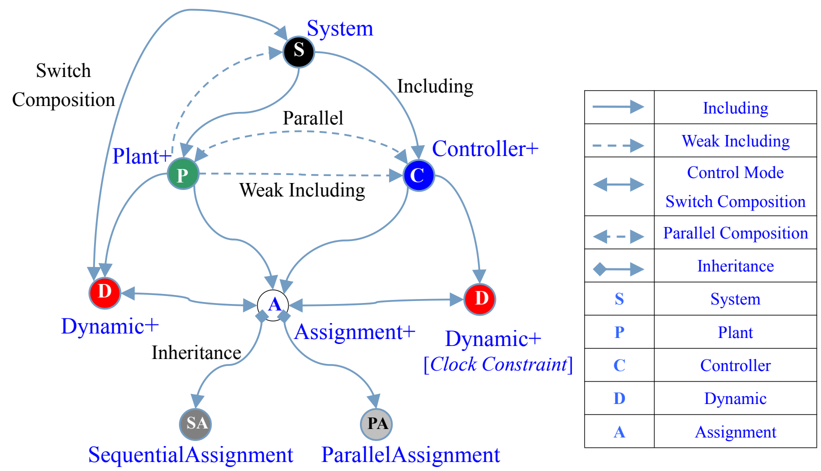


Fig. 1. The relationship of objects in *Apricot*

The *Continuous* method in a class that implements *Dynamic* has the following form:

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

where, for $1 \leq i \leq n, Var_i$ is a continuous variable. The natural number Nat_i represents the derivative order of Var_i over time. $MathExp_i$ is the mathematical expression with the definition: Let $Vars$ be the set of all variables of system, $\dot{V}ars$ denotes the set of derivative variables (various orders), e.g., if $v \in Vars$, then the first-order derivative $\dot{v} \in \dot{V}ars$ ($\dot{v}$ is represented by expression $dot(v, 1)$ in *Apricot*).

$$MathExp ::= Function(Vars, \dot{V}ars);$$

where, $Function$ is the mathematical function defined by the designer or one of the built-in functions in *Apricot*. Moreover, we also define the n-th order derivative over other variables, for example, $dot(x, y, n)$ is the n-th order derivative of x over y, i.e., $dot(x, y, n) = \frac{d^n x}{dy^n}$.

The *Invariant* statement specifies the properties of the system during the continuous evolution:

```
Invariant{
    MathExp_1 ∘ MathExp'_1 ;
        ...
    MathExp_n ∘ MathExp'_n ;
};
```

in which, $\circ \in$ {`==`,`<`,`<=`,`>=`,`>`,`!=`,`in`} is the relation operator. The operator '`in`' is used for intervals (or checking whether the control of an object is in a *Dynamic*). For example, '`t in [0,150]`' means that the value of variable t is in the interval `[0,150]`, i.e., `0` $\leq$ `t` $\leq$ `150`.

Interface *Assignment* has two sub-interfaces, *SequentialAssignment* and *ParallelAssignment*. The implementation of *ParallelAssignment* has a discrete method with the form:

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

If this discrete method is defined in a class that implements the interface *SequentialAssignment*, it is the sequential composition of n assignment statements. Otherwise, the parallel composition is the semantics that the assignment statements are supposed to be (in the case of interface *ParallelAssignment*). *SequentialAssignment* also supports traditional control structures, such as if-statement, for-loop, and method-call.

The *Composition* statement connects the *Dynamic* object and *Assignment* object by a *Condition* statement. The *Condition* statement has the form:

```
Condition{
    MathExp_1 ∘ MathExp'_1;
                ...
    MathExp_n ∘ MathExp'_n;
};
```

in which, $\circ \in \{$`==,<,<=,>=,>,!=,in`$\}$ is the relation operator.

3.2 Interface, Inheritance and Relationship

We define five primary built-in interfaces in *Apricot*. Each defines one key element of *Apricot*, and may contain method signatures, variable-requirements, constraint-indications and built-in block statements. From now on, these four parts are abbreviated as MVCB in this paper. Method signature defines the name and arguments of one method. Variable-requirement maintains the relations between the current interface and other interfaces. It also restricts the amount of objects. Constraint-indication demonstrates the limitation for the behavior of the object that implements the interface. The built-in block statement emphasizes the structure of the language, and indicates its proper application in a model. We illustrate the five interfaces as follows.

System Interface. Depicted as follows, where, in lines 2-3, '*Requires*' is a keyword for the declaration of variable-requirement, '1..∗' denotes the amount of entities is at least one. Therefore, each *System* object includes one or more *Plant* objects. The method signature '*Init*()' indicates that the *System* has an initializer without any argument nor value return (the type modifier is *void*). The names '*plants*' and '*controllers*' are the names indicating the variables of proper types (*Plant* and *Controller*), respectively.

```
1  interface System{
2      Requires plants[1..*] : Plant;
3      Requires controllers[1..*] : Controller;
4      void Init();
5  }
```

Plant Interface. The implementation of this interface includes several objects of type *Dynamic* and *Assignment*, and may have a subsystem (or controller) or not ([0..1] means zero or one). The *Composition* method is used for defining the composition relationships between *Dynamic* (or *System*) and *Assignment* objects. Each composition relationship takes three arguments: *source*, *action*, and *destination*. The *source* can be an object of type *Dynamic* or *System*, and, *action* is an object of type *Assignment*. The type of *destination* can be *Dynamic* or *System*. The composition relationship denotes the control switch from the source to the destination under the conditions defined in the *Condition* block statement. During the control switch the *Discrete* method of the *action* or the *Discrete* block is executed. If both (*action* and *Discrete* block) are present, the *Discrete* block would be executed first.

```
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{}; };
   }; }
```

where, '!' and '?' behind the composition name (*coms*) denote the asynchronous communication between compositions. For instance, *coms*(!) doesn't have to wait *coms*(?) to be valid. However, *coms*(?) has to wait *coms*(!). Thus, '!' indicates one kind of asynchronous message sending, which tells the proper compositions that are equipped with '?' can be executed. It is the case of synchronous communication when '?' and '!' are absent. The synchronous communication has the same semantics as the synchronization labels in hybrid automata. We can define the synchronization of two compositions A and B explicitly, as '$A \parallel B$'. And, '$A \sim B$' denotes asynchronous communication between A and B. In addition, independent composition is the case that, its name is followed only by the three arguments (i.e., *source*, *action*, and *destination*). The control switch defined by independent composition can be executed when the condition is satisfied.

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

```
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{}; };
    }; }
```

Dynamic Interface. It indicates that each implementation of *Dynamic* has a *Continuous* method and a built-in *Invariant* block statement. The method

'*Continuous*()' refers to the continuous evolution of the system states. The *Invariant* block is applied to define the evolution range of proper variables specified in the *Dynamic* object. 'Start()' is a built-in method, which indicates the starting of the continuous flow.

```
interface Dynamic{
    Continuous();
    Invariant{};
    Native void Start();
}
```

Assignment Interface. The *Assignment* interface has a '*Discrete*()' method. The *Discrete* method plays the role of the actions that would be executed during the control switch between dynamics. Moreover, two interfaces inherit the *Assignment* interface, i.e., *SequentialAssignment* and *ParallelAssignment*. The implementation of the former has the semantics of sequential composition for its assignment statements, the latter has a parallel composition semantics.

```
interface Assignment{
    void Discrete();
}
```

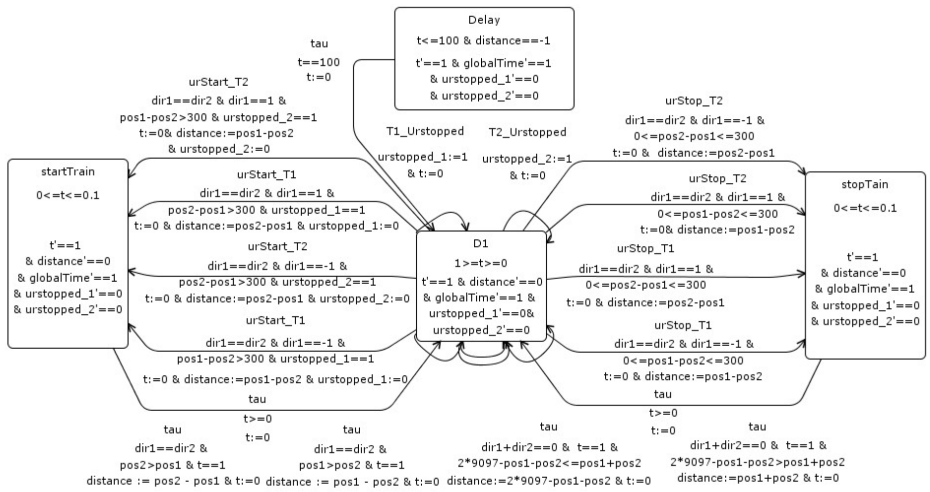
In addition, as the existence of MVCB, we claim that the inheritance of class or interface in *Apricot* should consider to inherit and follow the MVCB in the super-class or super-interface. Also, the implementation of interface in *Apricot* should take the MVCB into consideration.

4 Case Study: Subway Control system

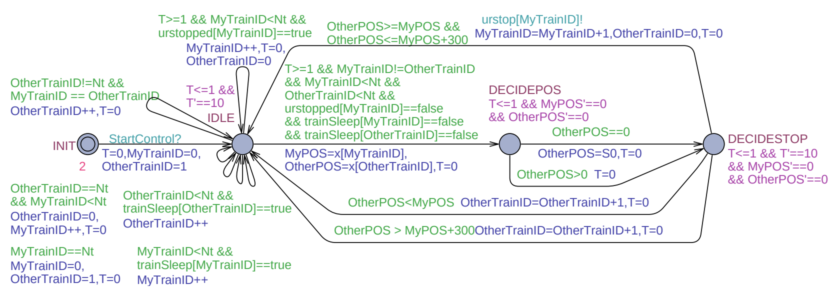
In this case, we consider the distance between trains in a subway control system. On the track of subway line, the train shall brake when its distance to the train ahead is less than or equal to 300 meters (urgent distance). We illustrate the controller of the urgent control for two trains (Fig. 2(a)) using hybrid automaton in SpaceEx. The controller is complex as an automaton. If we have more than two trains in the system, for example, 100 or 1000 trains, then more transitions would be added into the automaton. It is error-prone for one to maintain thousands of transitions in one automaton. Hybrid automata are not scalable for large systems. The hybrid automata in SpaceEx do not support array and control structures, UPPAAL [4] supports array, user-defined methods, and control structures, but it does not allow continuous variables in user-defined methods. The urgent control automaton (Fig. 2(b)) in UPPAAL is simpler than the automaton in SpaceEx, but it is still more complex than the model in *Apricot*.

In *Apricot*, the controller can be modeled much simpler than using hybrid automata, and scalable for the amount of trains. We show the control logic in Fig. 3(a), the class *Calculating* is used by the controller. The distance is calculated in the dual-for-statements[3] (Lines 11 to 33). We do not need to add more code for different amounts of trains. Moreover, in *Apricot*, the system dynamics declaration can be reused. The motivation of dynamics reuse is natural

[3] The for-statement and if-statement are similar to the respective statements in Java



(a) The controller of urgent distance control for two trains in SpaceEx



(b) The hybrid automaton for urgent control in UPPAAL

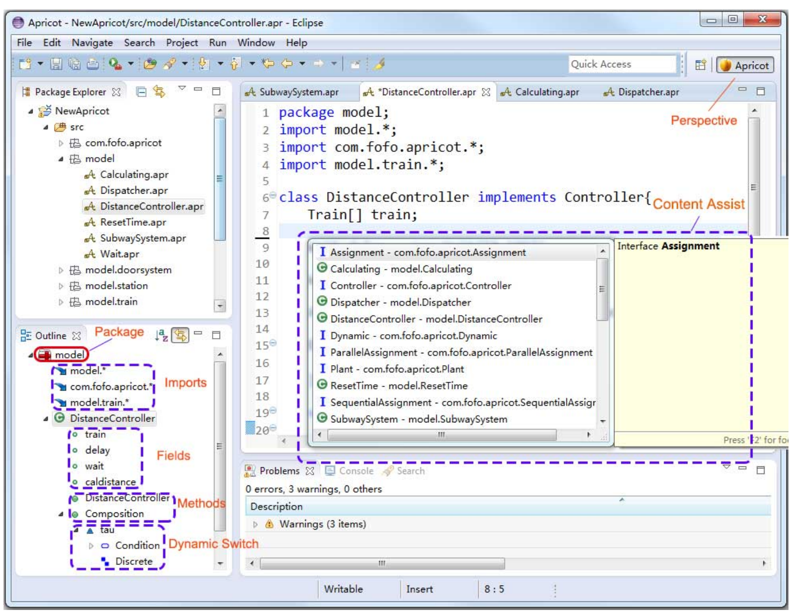
Fig. 2. Urgent Control with Hybrid Automata

and based on the structure of differential equations, Newton's laws and other dynamics laws (e.g., hydrodynamics). The laws of dynamics are usually stable on structure. With various values of parameters, the law can be applied in different scenarios. In *Apricot*, we allow the parameters in differential equations within *Dynamic* objects. Then, the *Dynamic* object can be employed in variant scenarios. This capability of our language is absent from the current notations or languages for hybrid systems.

In addition, in Matlab Simulink/Stateflow, if you want to reuse the same state or subchart many times across different charts or models to facilitate large-scale modeling, you can use atomic subcharts in 'discrete-time' charts. However, continuous-time charts do not support atomic subcharts. In *Apricot*, we can reuse the declarations of *Dynamic*, *Plant*, and *Controller*. As a result, in *Apricot*, we can reuse the declarations ranged from components to concrete dynamics, and assignments. The language is coarse-grained on components and fine-grained for dynamics and assignments.

```
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
```

(a) The calculation for distance between trains



(b) The modeling tool XtextApricot

Fig. 3. Model View of Apricot

For the modeling tool *XtextApricot*, we illustrate the view of class *DistanceController* in Fig. 3(b) with the *Apricot* perspective. In the outline view, one can navigate the imports, fields and methods. And, one can benefit from the content assist feature in the source view. For more details, we recommend the reader to visit the website of *XtextApricot* as we mentioned in Section 1.

5 Operational Semantics

Usually, structural operational semantics ([14], SOS) is applied to the programs and operations on discrete data. In order to deal with continuous data, we need to abstract the continuous features, and then obtain a discrete view of the continuous data for hybrid system.

5.1 Configurations

Any insight into a hybrid system is obtained through the states of the system. After the system start-up, it is always accompanied with a state at each time point. All the states compose one state space of the system. Based on the state space, we can check whether some specific states can be reached by the system from some proper initial states. In addition, to reveal the relation between statements and states, we also need to pay attention to the statements (control flow) throughout the system execution. These can be used to check the statement-related properties. For example, we can check that some particular methods are not reachable or executed by the system with the knowledge of both statement and state.

Definition 2 (Configurations). *We define the set of configurations with statements, states, and types, formally as follows:*

$$\begin{aligned}
\mathbf{C} &::= \langle \mathcal{P}(\Theta), \mathcal{P}(\Sigma), \mathcal{P}(\mathbf{T}) \rangle, \\
\Theta &::= \{\vartheta_1.\vartheta_2.\cdots.\vartheta_n \mid \vartheta_i \textit{ is a statement of Apricot}\}, \\
\Sigma &::= \mathbf{Vars} \times \mathbf{Vals}, \\
\mathbf{T} &::= \mathbf{Vars} \times \mathbf{Types},
\end{aligned}$$

where $1 \leq i \leq n$, Θ denotes the set of prefix annotated statements, $\mathcal{P}(\Theta)$ is the power set of Θ. Σ is the union of all functions that map from the set of variables **Vars** *to the set of values* **Vals**. **T** *is the union of all functions which relate each variable in* **Vars** *with a type in* **Types**.

A prefix annotated statement is a list that begins with a variable ϑ_1 which denotes the system and ends with ϑ_n the currently executed statement or the expression to be evaluated. Along the list there will be objects or methods. *Apricot* model comprises more than one component, these components are paralleled. The first element of a configuration is a subset of Θ, it consists of the parallel prefix annotated statements. Moreover, considering the nondeterminism feature of hybrid systems, the model of *Apricot* may consist of numerous prefix annotated statements. Thus all the possible runs of the model can be illustrated by a tree structure, and each branch may have a different state space.

5.2 Axioms and Rules

Consider a single statement θ, for $\{\mathcal{P}re.\theta\} \in \mathcal{P}(\Theta)$, $\sigma \in \mathcal{P}(\Sigma)$, and $\tau \in \mathcal{P}(\mathbf{T})$, then $\langle \{\mathcal{P}re.\theta\}, \sigma, \tau \rangle \in \mathbf{C}$. For simplicity, we take $\mathcal{P}re.\theta$ for $\{\mathcal{P}re.\theta\}$ in the semantics rules ($\mathcal{P}re$ is the prefix).

Evaluation of Variables. For $x \in \mathbf{Vars}$, x is a normal variable,

$$\langle \mathcal{P}re.x, \sigma, \tau \rangle \rightarrow \sigma(x) \qquad \text{[normal-variable]}$$

$$\langle \mathcal{P}re.dot(x,n), \sigma, \tau \rangle \rightarrow f^{(n)}(\sigma(now)) \qquad \text{[derivative-variable]}$$

where $n \in \mathbb{N}$, i.e., n is a natural number, and f is a solution of the differential equation: '$dot(x,n) == MathExp$', now is the variable that records the time passing after the dynamics started. $now \in [a,b]$, i.e., now is the time-point after the dynamics (continuous flow) of the differential equation started. a and b are the start and end time-points for the dynamics, respectively. $f^{(n)}$ is the n-th order derivative of variable x over time.
Assignment. For single assignment, sequential, and parallel assignments:

$$\langle \mathcal{P}re.(v = e), \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma', \tau \rangle \qquad \text{[single-assignment]}$$

$$\frac{\langle \mathcal{P}re.S_1, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma_1', \tau \rangle}{\langle \mathcal{P}re.(S_1; S_2), \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.S_2, \sigma_1', \tau \rangle} \quad \text{[sequential-assignment]}$$

$$\frac{\begin{array}{c}\langle \mathcal{P}re.S_1, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma_1', \tau \rangle,\\ \langle \mathcal{P}re.S_2, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma_2', \tau \rangle\end{array}}{\langle \mathcal{P}re.(S_1 || S_2), \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma_3', \tau \rangle} \quad \text{[parallel-assignment]}$$

where, v is a variable, e is an expression, the updated state $\sigma' = \sigma[v \mapsto \sigma(e)]$. For sequential and parallel assignments, consider the assignment statements in the *Discrete* method S:

`void Discrete(){ x = y; y = x; }.`

(i) As Sequential Assignment: executing S in an initial state with $x = 0$ and $y = 1$, x and y are both evaluate to the value 1.
(ii) As Parallel Assignment: executing S in the same initial state, x and y exchange their values, x is changed to 1, y is 0. For assignment statements S_1 and S_2, v_1 and v_2 are the variables modified by S_1 and S_2, respectively, then $\sigma_3' = \sigma[v_1 \mapsto \sigma_1'(v_1), v_2 \mapsto \sigma_2'(v_2)]$, '$||$' denotes that the assignments (S_1 and S_2) in *Discrete* method of *ParallelAssignment* object are executed in parallel.

Dynamic. In *Apricot*, the *Dynamic* object consists of one *Continuous* method and an *Invariant* block. If the dynamic flow reaches the border of the *Invariant* and all the condition blocks of related switch compositions cannot be satisfied, the control will be waiting at the border.

$$\langle \mathcal{P}re.D_e, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.D_e, \sigma', \tau \rangle \qquad \text{[differential-equation]}$$

$$\frac{\begin{array}{c}\forall c \in C_s, \langle \mathcal{P}re.c, \sigma, \tau \rangle \rightarrow False,\\ \langle \mathcal{P}re.D_e, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.D_e, \sigma', \tau \rangle,\\ \exists i \in Inv, \langle \mathcal{P}re.i, \sigma', \tau \rangle \rightarrow False\end{array}}{\langle \mathcal{P}re.D_e, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.(dot(tw,1) == 1), \sigma, \tau \rangle} \quad \text{[termination-flow]}$$

(i) Differential Equation D_e of Continuous Variable v. Suppose that there exists a function $f : I \to \mathbb{R}$, I is a time-interval $[a, b]$, the value of v at time-point $t \in [a, b]$ is $f(t)$. Here, the continuous evolution following D_e starts at time a. The end point b is for some proper time-point greater than or equals a. Thus, there exists one time-point $t \in [a, b]$, $\sigma' = \sigma[v \mapsto f(t)]$. We call f the *Real-Function* for D_e, and t the *Proper-Time*.

(ii) Termination of Flow. When the dynamic flow reaches the border of the *Invariant* and no valid composition relationship exists, the control will be waiting at the border if any forward flow would violate the *Invariant*. The set C_s is the *Condition* blocks related to the current *Dynamic* object. And, Inv is the *Invariant* block in the *Dynamic* object. During the continuous evolution, the conditions in Inv should be satisfied all the time. For $\forall t \in (a, b]$, $\sigma' = \sigma[v \mapsto f(t)]$, in which, $f : I \to \mathbb{R}$, $I = [a, b]$, $f(t)$ is the value of v at the time-point $t \in I$. And, tw is a built-in variable, recording the waiting time after the flow terminated.

Method Invocation. For method m and different kinds of arguments,

$$\langle \mathcal{P}re.m(), \sigma, \tau\rangle \to \langle \mathcal{P}re'_1.S, \sigma, \tau\rangle \qquad \text{[zero-ary]}$$

$$\langle \mathcal{P}re.m(exp[1..n]), \sigma, \tau\rangle \to \langle \mathcal{P}re'_2.S, \sigma', \tau'\rangle \qquad \text{[fixed-ary]}$$

where, $\mathcal{P}re'_1 = \mathcal{P}re.m$ and S is the body of method m, $\mathcal{P}re'_2 = \mathcal{P}re.m(exp[1..n])$. For $1 \leq i \leq n$, $arg[i]$ is a new variable, $\sigma' = \sigma[arg[i] \mapsto \sigma(exp[i])]$. If $\tau(exp[i])$ is a subtype of the defined type of $arg[i]$, then $\tau' = \tau[arg[i] \mapsto \tau(exp[i])]$. Otherwise, $\tau'(arg[i])$ takes the defined type of the formal parameter.

Start Dynamics. For *Dynamics* D_1 and D_2,

$$\frac{\langle \mathcal{P}re.D_1, \sigma, \tau\rangle \to \langle \mathcal{P}re.D_1, \sigma_1, \tau\rangle,\quad \langle \mathcal{P}re.D_2, \sigma, \tau\rangle \to \langle \mathcal{P}re.D_2, \sigma_2, \tau\rangle}{\langle \mathcal{P}re.(D_1||D_2), \sigma, \tau\rangle \to \langle \mathcal{P}re.(D_1||D_2), \sigma', \tau\rangle} \qquad \text{[start-dynamics]}$$

The composition for start statements, is the parallel evolution of the continuous flows, let $D_1||D_2 \equiv D_1.Start(); D_2.Start()$, then, $\sigma_1 = \sigma[v_1 \mapsto f_1(t)]$, $\sigma_2 = \sigma[v_2 \mapsto f_2(t)]$. The new state, $\sigma' = \sigma_1[v_2 \mapsto f_2(t)] = \sigma_2[v_1 \mapsto f_1(t)] = \sigma[v_1 \mapsto f_1(t), v_2 \mapsto f_2(t)]$. Here, f_1 and f_2 are the *Real-Functions* for D_1 and D_2, respectively. And, t is the *Proper-Time*.

Composition Relationship (Local). Let D_1 and D_2 represent two *Dynamic* objects, they may be the same object. Let C be one of the *Condition* blocks related to D_1 and D_2. For *Composition Relationship* CR and the corresponding *Assignment* object A, let R be the name of the *Composition Relationship*, then $CR \equiv R(D_1, A, D_2)\{C\}$. For convenience, we simplify it to $CR \equiv R(D_1, A, D_2, C)$. Thus, we have the valid composition relationship,

$$\frac{\langle \mathcal{P}re.C, \sigma, \tau\rangle \to True,\quad \langle \mathcal{P}re.A, \sigma, \tau\rangle \to \langle \mathcal{P}re.Skip, \sigma', \tau\rangle,\quad \langle \mathcal{P}re.D_2.Inv, \sigma', \tau\rangle \to True}{\langle \mathcal{P}re.R(D_1, A, D_2, C), \sigma, \tau\rangle \to True} \qquad \text{[valid-composition]}$$

where, Inv is the *Invariant* of D_2. And, the control switch from D_1 to D_2 may occur when the relationship is valid,

$$\frac{\langle \mathcal{P}re.R(D_1, A, D_2, C), \sigma, \tau\rangle \to True}{\langle \mathcal{P}re.D_1, \sigma, \tau\rangle \to \langle \mathcal{P}re.D_2, \sigma', \tau\rangle} \qquad \text{[dynamic-switch]}$$

Note that, the control switch may not take place even though the relationship is valid. It means that, if the *Invariant* of D_1 is true and D_1 can continue the continuous evolution without violating the *Invariant*, then the choice to switch or continue the flow itself is nondeterministic.

Parallel-Composition (Global). The parallel composition for *Composition Relationship* consists of two categories, synchronization and asynchronization.

(i) Synchronization: For two composition relationships CR_s and CR_t, $CR_s \equiv R_s()(D_{s_1}, A_s, D_{s_2}, C_s)$, $CR_t \equiv R_t()(D_{t_1}, A_t, D_{t_2}, C_t)$, the parallel composition relationship is defined as $CR_s \mid\mid CR_t$. The parallel composition relationship is valid as follow,

$$\frac{\begin{array}{c}\langle \mathcal{P}re.(CR_s \ and \ CR_t), \sigma, \tau\rangle \rightarrow True, \\ \langle \mathcal{P}re.(A_s||A_t), \sigma, \tau\rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma', \tau\rangle, \\ \langle \mathcal{P}re.(D_{s_2}.Inv \ and \ D_{t_2}.Inv), \sigma', \tau\rangle \rightarrow True\end{array}}{\langle \mathcal{P}re.(CR_s||CR_t), \sigma, \tau\rangle \rightarrow True} \quad \text{[valid-synchronization]}$$

where, the Boolean operator '*and*' represents the conjunction relation. The rule for synchronized parallel composition is:

$$\frac{\langle \mathcal{P}re.(CR_s||CR_t), \sigma, \tau\rangle \rightarrow True}{\langle \mathcal{P}re.(D_{s_1}||D_{t_1}), \sigma, \tau\rangle \rightarrow \langle \mathcal{P}re.(D_{s_2}||D_{t_2}), \sigma', \tau\rangle} \quad \text{[synchronization]}$$

(ii) Asynchronization: For two composition relationships CR_s and CR_t, $CR_s \equiv R_s(!)(D_{s_1}, A_s, D_{s_2}, C_s)$, $CR_t \equiv R_t(?)(D_{t_1}, A_t, D_{t_2}, C_t)$, the parallel composition relationship is defined as follows:

$$CR_s \ \sim \ CR_t \equiv R_s(!)(D_{s_1}, A_s, D_{s_2}, C_s) \ \sim \ R_t(?)(D_{t_1}, A_t, D_{t_2}, C_t).$$

CR_s has the meaning of sending, CR_t is receiving when CR_s is sending. Here, CR_s does not synchronize with CR_t. But CR_t has to wait CR_s. If both CR_s and CR_t are valid at the same time, then,

$$CR_s \ \sim \ CR_t \equiv R_s()(D_{s_1}, A_s, D_{s_2}, C_s) \mid\mid R_t()(D_{t_1}, A_t, D_{t_2}, C_t).$$

Otherwise, if only CR_s is valid, then the control switch of CR_s is executed independently. As a result, we have the rules:

$$\frac{\langle \mathcal{P}re.R_s(!)(D_{s_1}, A_s, D_{s_2}, C_s), \sigma, \tau\rangle \rightarrow True}{\langle \mathcal{P}re.D_{s_1}, \sigma, \tau\rangle \rightarrow \langle \mathcal{P}re.D_{s_2}, \sigma', \tau\rangle} \quad \text{[asynchronized-sending]}$$

$$\frac{\begin{array}{c}\langle \mathcal{P}re.R_s(!)(D_{s_1}, A_s, D_{s_2}, C_s), \sigma, \tau\rangle \rightarrow True, \\ \langle \mathcal{P}re.R_s(?)(D_{s_1}, A_s, D_{s_2}, C_s), \sigma, \tau\rangle \rightarrow True\end{array}}{\langle \mathcal{P}re.(CR_S \sim CR_t), \sigma, \tau\rangle \rightarrow \langle \mathcal{P}re.(CR_S||CR_t), \sigma, \tau\rangle} \quad \text{[conditional-synchronization]}$$

System-Initialization. The method `System.Init()` consists of three tasks. The first one is the initialization for variables. The second is *Composition Relationship* (abbreviated as *CR*) registration for various *Dynamic* objects. The last is the dynamics starting.

```
void Init(){
        Initialize(Variables);          // task-1
        Register(Compositions);         // task-2
        Start(Dynamics);                // task-3
}
```

The task-1 `Initialize`(**Variables**) can be done by assignment statements as discussed previously. task-3 `Start`(**Dynamics**) is also illustrated in the semantics of **Start Dynamics**. The registration of *CR* binds the *CR* to *Dynamic* objects. For example, let the *Composition* method in *Plant Pt* be:

```
void Composition(){
    CR(D_1,A,D_2){C};
}
```

then, the method call 'Pt.`Composition`()' would register CR (with *Condition* C, *Assignment* A and the destination *Dynamic* object D_2) onto the source *Dynamic* object D_1. During the flow evolution of D_1, the system monitors the CR all the time, and may take the control switch from D_1 to D_2 when the *Condition* C is true.

$$\langle \mathcal{P}re.Register(CR), \sigma, \tau\rangle \to \langle \mathcal{P}re.Skip, \sigma', \tau\rangle \qquad \text{[CR-registration]}$$

where, $\sigma' = \sigma[D_1.CRS \mapsto D_1.CRS + CR]$, the new CR is appended to the CR list of the *Dynamic* object D_1.

6 Related Work

The language of hybrid automata is a graphical modeling language on hybrid systems. Formal verification tools for hybrid automata usually have a textual language to support the description on the behavior of complex systems (e.g., HyTech [11], d/dt [3], PHAVer [7], ProHVer [17], HYSDEL [15]). Moreover, the de facto industrial standard simulation toolset Matlab Simulink/Stateflow also has a C-like language to support textual description of complex system behavior. In addition, the automated and interactive theorem prover KeYmaera [13] for hybrid systems took the textual program notation *hybrid program* as the input for the tool. In this paper, we propose the textual notation *Apricot* which is capable of modeling large-scale hybrid systems.

Modelica [9] is a multi-domain object-oriented modeling language, it involves systems relating electrical, mechanical, control, and thermal components. The semantics of Modelica is prone to be deterministic, however, in the area of hybrid systems, it is needed and important to consider the non-deterministic evolution behaviors of the system. CHARON [2] is a modular specification language for hybrid systems. The mode in CHARON is a hierarchical state machine, and can be shared in multiple contexts. However, the features of inheritance and dynamic instance creation are not supported in CHARON.

7 Conclusion and Future Work

In this paper, we have proposed an object-oriented language for modeling hybrid systems. We described the syntax and operational semantics of *Apricot* in detail. The language combines the features from DSL and OOL, that fills the gap between design and implementation. In addition, we have a prototype tool for modeling hybrid systems with *Apricot*.

As the design targets in hybrid systems are the real physical objects in our environment, the concept of object is natural, and fits well for hybrid systems.

The object-oriented feature of *Apricot* makes the relationship between components of a system more clear, and also supports code reuse, inheritance and composition. The code reuse is the first step to produce reusable components, thus the design of large-scale systems can benefit from this. The second is inheritance, it is the way to build complex systems from simpler ones. Because, inheritance allows designers to preserve or modify object's original behavior, or append new behavior, making the system more and more complex. The third is the composition, it constructs the relationship between objects, components and subsystems, making the system model scalable and distinct for implementation.

For the future work, an important direction is the formal verification for *Apricot* models. As the non-linear dynamics are acceptable in our language, we need to propose an efficient abstract approach for the non-linear dynamics. Such abstraction can be achieved by adopting abstract interpretation [5]. For the feature of dynamic object instantiation, it can be reconfigured into a special form of uncertainty of system behavior, then verified by traditional verification techniques. Another future work is the simulation of *Apricot* models. One feasible way to achieve this is to translate *Apricot* models into statecharts in Matlab Simulink/Stateflow. This translation can be conveniently implemented within the Xtext framework.

References

1. Alur, R., Courcoubetis, C., Henzinger, T., Ho, P.: Hybrid automata: An algorithmic approach to the specification and analysis of hybrid systems. In: Hybrid Systems, LNCS, vol. 736, pp. 209–229. Springer-Verlag (1993)
2. Alur, R., Dang, T., Esposito, J., Hur, Y., Ivancic, F., Kumar, V., Mishra, P., Pappas, G., Sokolsky, O.: Hierarchical modeling and analysis of embedded systems. Proceedings of the IEEE 91(1), 11–28 (2003)
3. Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: Proceedings of CAV'02, LNCS, vol. 2404, pp. 365–370. Springer-Verlag (2002)
4. Behrmann, G., David, A., Larsen, K., Hakansson, J., Petterson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: Proceedings of QEST. pp. 125–126 (2006)
5. Cousot, P., Cousot, R., Mauborgne, L.: Theories, solvers and static analysis by abstract interpretation. Journal of the ACM 59(6), 31:1–31:56 (2012)
6. Fowler, M.: Domain-specific languages. Addison-Wesley Professional (2010)
7. Frehse, G.: Phaver: algorithmic verification of hybrid systems past hytech. International Journal on Software Tools for Technology Transfer 10(3), 263–279 (2008)
8. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proceedings of CAV'11. LNCS, vol. 6806, pp. 379–395. Springer-Verlag (2011)
9. Fritzson, P., Engelson, V.: Modelica – a unified object-oriented language for system modeling and simulation. In: Proceedings of ECOOP'98, LNCS, vol. 1445, pp. 67–90. Springer-Verlag (1998)
10. He, J.: From csp to hybrid systems. In: A Classical Mind, Essays in Honour of C.A.R. Hoare, pp. 171–189. Prentice Hall International (1994)
11. Henzinger, T., Ho, P., Wong-Toi, H.: Hytech: A model checker for hybrid systems. International Journal on Software Tools for Technology Transfer 1(1), 110–122 (1997)

12. Platzer, A.: Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics. Springer-Verlag (2010)
13. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR. LNCS, vol. 5195, pp. 171–178. Springer-Verlag (2008)
14. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Department of Computer Science, Aarhus university (September 1981)
15. Torrisi, F.D., Bemporad, A.: Hysdel-a tool for generating computational hybrid models for analysis and synthesis problems. IEEE Transactions on Control Systems Technology 12(2), 235–249 (2004)
16. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
17. Zhang, L., She, Z., Ratschan, S., Hermanns, H., Hahn, E.: Safety verification for probabilistic hybrid systems. In: Proceedings of CAV'10, LNCS, vol. 6174, pp. 196–211. Springer-Verlag (2010)
18. Zhou, C., Wang, J., Ravn., A.P.: A formal description of hybrid systems. In: Hybrid Systems III, LNCS, vol. 1066, pp. 511–530. Springer-Verlag (1996)