An Object-Oriented Language for Modeling of Hybrid Systems

Huixing Fang*, Huibiao Zhu*, Jianqi Shi†
*Shanghai Key Laboratory of Trustworthy Computing,
Software Engineering Institute, East China Normal University, Shanghai, China
†School of Computing, National University of Singapore, Singapore
Email: {wxfang, hbzhu}@sei.ecnu.edu.cn, shijq@comp.nus.edu.sg

***Abstract*—Hybrid systems arise in embedded control from the interaction between continuous physical behavior and discrete digital controllers. In this paper, we propose *Apricot* as a novel object-oriented language for modeling hybrid systems. The language takes the advantages of domain-specific and object-oriented languages, which fills the gap between the design and implementation. With respect to the application of Apricot, we demonstrate 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 (a plug-in for Eclipse) for our proposed 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 many key features.**

I. INTRODUCTION

The Hybrid system consists of discrete control programs and continuous physical behaviours, such system combines logical decision making with the generation of continuous-valued control laws. Usually, hybrid systems modelled as finite state machines are coupled with partial or ordinary differential equations, and difference equations. In order to model hybrid systems, numerous modelling approaches have been proposed: hybrid automata [1], Hybrid CSP [2], [3], HyPA [4] and hybrid programs [5]. With respect to the formal verification of hybrid systems, various tools can be used, for instance, HyTech [6], d/dt [7], PHAVer [8], ProHVer [9], SpaceEx [10], and KeYmaera [11]. The common feature of these works is that most of them focus on the high-level abstraction of hybrid systems. However, both abstraction and easy usability are important in industry. Therefore, there is a demand for developing a modelling language that caters for these two concerns for hybrid systems. Nowadays, automata, process algebras and state-charts are the three most used notations for modelling hybrid systems.

Firstly, automata based notations are not the best choice 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 only 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 usually thousands of objects, such as 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 fit for hybrid systems as they take the advantage of solid theoretical foundation of formal analysis. However, in industry, they are not accepted widely by designers and developers due to the complicated symbols, mathematical abstractions and various concept abbreviations comprised in the models. Expert help and guidance are often required by industrial practitioners in order to apply such formalisms even for some simple problems.

Thirdly, in practical usage scenario, applying state-chart based notations is a de facto approach for model-based development of embedded systems. It is analogous to automata-based notations, whereas 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. Nonetheless, the state-chart approach is weak on model reuse as well. For instance, the continuous-time chart in Matlab Simulink/Stateflow cannot be reused. In addition, inheritance of components is likewise hard to be implemented in this scenario.

Apricot is a modelling language that has a clear and simple syntax, an explicit semantics, and an appropriate architecture for constructing models for hybrid systems. 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, they have the ability to describe the ownership among different objects. The constraint-indications denote the requirement assumptions that the behaviour of the system is forced to conform. The built-in block statements denote the constraints on the model design and the position that the statement should be in. As a consequence, our approach firstly enhances and clarifies the relationship of various components by variable-requirements, specifies the limitation of components by constraint-indications in the next place, then 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 of Domain-Specific (DSL [12],

†Corresponding author.

[13]) and Object-Oriented Languages (OOL). The DSL notations employed in Apricot are appropriate for building component skeleton. As a result, it makes developers easier to interact with domain experts during the design phase. On the other hand, OOL is more friendly to developers in industry. The combination of DSL and OOL in Apricot fills the gap between the designs in abstract level and the implementation in concrete level.

(iii) For tool support, we implemented the IDE of Apricot language as a Eclipse plugin with Xtext[1] , and developed a prototype tool *XtextApricot*[2] for modelling the hybrid system. The prototype tool is now only designed for modelling and analysing the hybrid system. Formal verification is planed in the next version.

The language of hybrid automata is a graphical modelling language on hybrid systems. Formal verification tools for hybrid automata usually have a textual language to support the description on the behaviour of complex systems (e.g., HyTech [6], d/dt [7], PHAVer [8], ProHVer [9], HYSDEL [14]). Moreover, the de facto industrial standard simulation toolset Matlab Simulink/Stateflow also has a C-like language to support textual description of complex system behaviour. In addition, the automated and interactive theorem prover KeYmaera [11] 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 modelling large-scale hybrid systems.

Modelica [15] is a multi-domain object-oriented modelling 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 necessary and important to consider the non-deterministic evolution behaviours of the system. CHARON [16] 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.

This paper is organised as follows. Section II is an overview of hybrid systems. We describe the sample case and the features of Apricot in Section III. In Section IV, we propose the syntax of Apricot. Section V presents our case study: a subway control system. The operational semantics is described in Section VI. Finally, we draw our conclusions in Section VII.

II. Background of Hybrid Systems

Hybrid systems consist of discrete control programs and continuous physical behaviour, such systems exhibit both continuous and discrete dynamic behaviour. We illustrate the concept of hybrid systems by listing the below definitions of hybrid automata.

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

[2]The tool and complete model for the subway control system are available at http://www.apricotresearch.com/

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 $m, n \in M$, if HA has a transition from m to n, then $(m, n) \in E$ is a control switch. E is the set of all the control switches.
- 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.
- Σ is a finite set of events.
- Synchronisation labels. For each control switch $e \in E$, the corresponding synchronisation 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, where $1 \leq i \leq n$.

Example 1: In Fig. 1, the hybrid automaton contains only one control mode (with the name 'IDLE'). The set of variables is $\{H, V, c, g\}$. Variable H represents the height of the ball, V denotes the velocity, c is a physical value that affects the ball's velocity when it rebounds from the ground. In addition, variable g indicates the acceleration of gravity on earth. Initially, the height of the ball is 10 meters above the ground with velocity of 0 m/s. When we loosen the ball, it starts falling, the dynamics of the ball follows the flow condition: $\dot{H} = V, \dot{V} = -g$ (i.e., V and $-g$ are the first-order derivatives of H and V over time, respectively). Moreover, the invariant ($H \geq 0$) is respected by the ball during the falling. The invariant ensures the ball never drops into the ground. In the hybrid automaton, the only one control switch is from the mode (IDLE) to itself. The jump condition describes the precondition ($H = 0$) and postcondition ($V' = -cV$) for the switch. It means that, if the height of the ball is 0 m, then the switch can be taken, and the new value for the velocity equals $-cV$ (i.e., multiply V by $-c$ is the new value of V).

Definition 2 (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 $HA_2 = \langle V_2, M_2, F_2, I_2, \eta_2, E_2, J_2, \Sigma_2, \sigma_2\rangle$. 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, $M = M_1 \times M_2$,

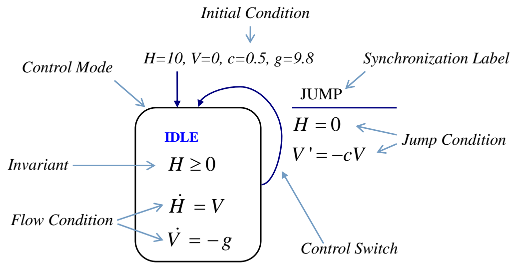


Fig. 1. Hybrid automaton of a bouncing ball

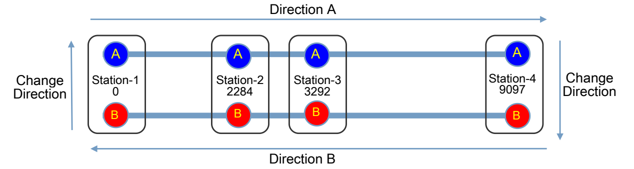


Fig. 2. The subway (metro) line, train running path and stations. For each station, platform-A is depicted as a solid blue circle, and platform-B the solid red circle. The positions (or coordinates) of stations are 0, 2284, 3292, and 9097, respectively. As a result, the distance between two stations can be calculated, for instance, the distance between Station-1 and Station-2 is 2284 meters

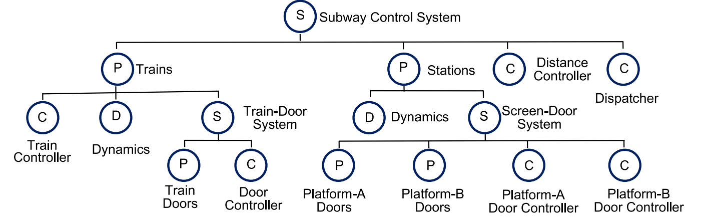


Fig. 3. A hierarchical structure of the subway control system. The capital letters 'S', 'P', 'C', 'D' denote the entities of system, plant, controller, and dynamic, respectively

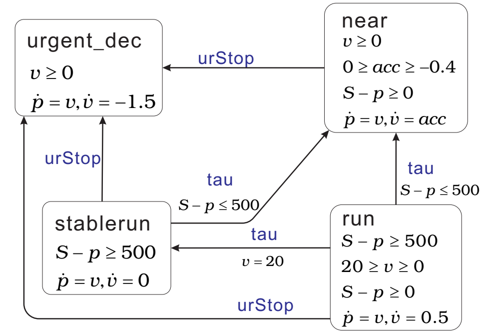


Fig. 4. Some parts of the hybrid automaton. Variable S denotes the position of the station in front of the train. The position of the train is represented by variable p, the velocity is by v. Since $0 \geq acc \geq -0.4$ (m/s^2), variable acc denotes the deceleration of the train in the mode $near$

$V = V_1 \cup V_2$, $\sigma = \sigma_1 \cup \sigma_2$. 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 synchronisation label $\sigma(e)$, then e_1 and e_2 should be synchronised in $HA_{1||2}$.

III. The Running Case and Motivation of Apricot

In our case study of a subway control system, we have four trains and four subway stations. In Fig. 2, the train runs in clockwise direction along the metro line, that is from Station-1 (first station) to Station-4 (final station), then from Station-4 to Station-1, and so on. The train reaches the final station in direction A, and stops at platform-A of the station. After several minutes (the time for picking up passengers), it will leave from platform-A and go to platform-B. Its direction will be changed from direction A to direction B, then it stops at platform-B. Similarly, at the first station, the train will change the direction from B to A if it runs from platform-B to platform-A.

Fig. 3 illustrates part of the hierarchical structure of the subway control system in our case study. Trains and stations are the plants of the system. The distance controller is designed for the trains that they should not be too close during running. Initially, all trains stop at the first station waiting to be scheduled. The scheduling is managed by the dispatcher controller. For each train, it contains a controller to control the start, stop of the train and communicate with the screen door system when it stops at a station. In addition, the train contains, some dynamics represent its continuous behaviors, and a sub-system of train doors (i.e., Train-Door system). This Train-Door system consist of the plants of train doors and a corresponding controller to manage their behaviors such as open and close.

The model of the train in our case can be modeled in a hybrid automaton, it only contains 18 modes, but it is not small as a graph. Thus, in Fig. 4, we only show some parts of the automaton instead of a complete one. We have four modes in Fig. 4, $urgent_dec$ is the mode of emergency deceleration when the distance between the train and its front train is less than or equal to the urgent distance (e.g., 300 meters in our case). The deceleration is $-1.5m/s^2$ in the mode $urgent_dec$. The mode run is the normal acceleration of the train, it will switch to mode $stablerun$ when the velocity reaches $20m/s$, then the train preserves the velocity at $20m/s$. The train will switch to mode $near$ when its distance to the front station is less than or equal to 500 meters.

Analysis of hybrid automaton: All the modes in Fig. 4 have similar flow conditions. They can be described as the form,

$$\dot{p} = v; \;\; \dot{v} = acc.$$

Therefore, the continuous behavior of the train can be modeled only in one template declaration, for instance, a class in Apricot. Then, all the modes of the train can reuse this template declaration with specific values of the variable acc. Moreover, the declaration of the synchronization label and jump condition in control switches can be shared between different modes as well. For example, the control switch $e_1 = (stablerun, near)$ and $e_2 = (run, near)$ have the same synchronization label and jump condition, that is $\sigma(e_1) = \sigma(e_2) = tau$ and $J(e_1) = J(e_2) = (S - p \leq 500)$. Unfortunately, these two kinds of reuse of declarations are not supported in hybrid automata. In our language, we can benefit from code reuse for these scenarios.

```
1 class TrainBehavior implements Dynamic{
2  Real p;// position
3  Real v;// velocity
4  Interval acc;// acceleration
5  real pos1; //The lower bound of the position
6  real pos2; //The upper bound of the position
7  real vel1; //The lower bound of the velocity
8  real vel2; //The upper bound of the velocity
9  TrainBehavior(Real p, Real v, Interval acc,
10   real pos1, real pos2, real vel1, real vel2){
11   this.p = p; this.v = v; this.acc = acc;
12   this.pos1 = pos1; this.pos2 = pos2;
13   this.vel1 = vel1; this.vel2 = vel2;
14  }
15 ...//Some extra codes
16  void Continuous(){
17  //Declares the continuous behavior
18      dot(p,1) == v;
19      dot(v,1) == acc;
20  }
21  Invariant{
22    p in [pos1, pos2];
23    v in [vel1, vel2];
24  };
25 }
```

Fig. 5. The declaration of continuous behavior of the train in Apricot. The interface *Dynamic* is a built-in interface of our language for continuous behaviors. The expression 'p in $[pos1, pos2]$' indicates the range of p, that is $pos1 \leq p \leq pos2$

The motivation of our language is focus on the code reuse for different levels of the system declarations. In our language, the reuse of declarations ranged over continuous behaviors, discrete behaviors, plants, controllers, and system declarations. As an example, the continuous behavior of the train can be declared in Apricot as illustrated in Fig. 5. The differential equations in the method 'Continuous()' describe the flow condition of the train. The equation 'dot(p,1)==v' represents that the first-order derivative of p is equal to v, in which p and v denote the position and velocity of the train, respectively. Now, the mode *run* in Fig. 4 can be declared as the instance creation statement in our language: `TrainBehavior run = new TrainBehavior(p,v,[0.5,0.5],-Inf,S-500,0,20)`. The expressions `-Inf` and `Inf` denote the negative infinity and positive infinity respectively. The interval `[0.5,0.5]` is a value of type `Interval` . Moreover, the mode *near* can be instantiated in our language as: `TrainBehavior near = new TrainBehavior(p,v,[-0.4,0],-Inf,S,0,Inf)`. Also, `[-0.4,0]` is a value of type `Interval`, it indicates the range of values for variable *acc* in the mode *near*.

As the control switches from *run* and *stablerun* to *near* have the same synchronization label and jump condition in the hybrid automaton in Fig. 4. In our language, the declaration of the synchronization label and jump condition in control switches can be shared as follows:

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

in which, `tau({run,stablerun},,near)` defines the control switches with `tau` as the name, and three parameters.

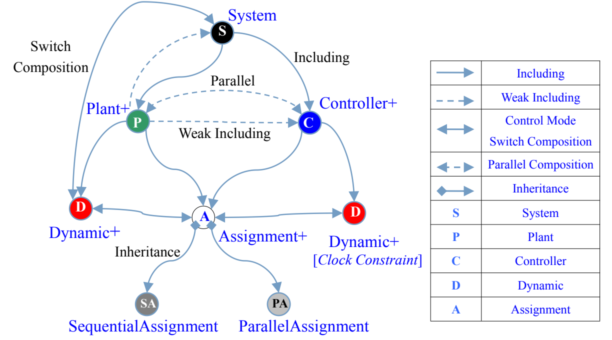


Fig. 6. The relationship of objects in Apricot

The first parameter {`run, stablerun`} represents the set of modes that are the sources of these two switches. The second parameter is for a discrete assignment object, but is ignored here. The third parameter `near` indicates the destination of these two control switches. Theses two control switches share the same jump condition `abs(S-position) <= 500`, it is positioned in the '`Condition`' block as described above. The expression `abs(e)` represents the built-in mathematical function for calculating the absolute value of `e`.

For higher level code reuse, in our case, the declaration of train doors in our language also can be used for screen doors directly. The declaration of corresponding controller for train doors is suitable for screen doors as well. The case of subway control system in this paper is much more simpler than the case in practice such as Shanghai Metro or New York City Subway. Without proper code reuse, the modeling of a complex system similar to our case is not a simple process with hybrid automata. Various levels of code reuse and a proper hierarchical architecture (as described in Sect. IV) of system are the basic concern for the modeling of complex hybrid systems in practice.

IV. 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.

A. Architecture and Fundamental Syntax

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

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.

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 switch composition under the condition in '$Condition^{+}$'. In this paper, the control 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. In Fig. 5, the differential equations in the method $Continuous$ of class $TrainBehavior$ describe the continuous flow constraints of a train. 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. For example, a train usually adjusts to the right position when it nearly stops at a subway station, we abstract this kind of physical behavior as an assignment, e.g., $p = 0$ whenever 0 is the right position and the distance between train and the right position at the station is $0.5\ m$.

In Fig. 6, 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 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. The *Continuous* method in a class that implements *Dynamic* has the following form:

```
1  void Continuous(){
2      dot(Var_1, Nat_1) == MathExp_1;
3                ...
4      dot(Var_n, Nat_n) == MathExp_n;
5  }
```

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 our language).

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

where, $Function$ is the mathematical function defined by the designer or one of the built-in functions in Apricot. We illustrate several built-in functions of Apricot in Table I. 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:

TABLE I. BUILT-IN FUNCTIONS

Function Form	Meaning
$round(x)$	Round x to the nearest integer, omitting decimal fractions smaller than 0.5, e.g. $round(2.5) = 3$, $round(0.4) = 0$.
$div(x, y)$	Truncated division, quotient rounded towards zero, and the remainder therefore has the same sign as the dividend. For example, $div(-5, 2) = -2$, and $div(5, 2) = 2$.
$gcd(x_1, ..., x_n)$	Greatest common divisor of x_1, x_2, ..., x_n with sign matching x_1.
$abs(x)$	Returns a positive value with the magnitude of x.
$gamma(x)$	The gamma function at positive integer x. For example, $gamma(5) = 5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

```
1  Invariant{
2      MathExp_1 ∘ MathExp'_1;
3                ...
4      MathExp_n ∘ MathExp'_n;
5  };
```

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., $0 \leq \text{t} \leq 150$.

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

```
1  void Discrete(){
2      Variable_1 = MathExp_1;
3                ...
4      Variable_n = MathExp_n;
5  }
```

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:

```
1  Condition{
2      MathExp_1 ∘ MathExp'_1;
3                ...
4      MathExp_n ∘ MathExp'_n;
5  };
```

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

B. 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.

```
1  interface Plant{
2    Requires dy[1..*] : Dynamic;
3    Requires ass[1..*] : Assignment;
4    Requires sy[0..1] : System;
5    Requires controller[0..1] : Controller;
6    void Composition(){
7      Requires coms[1..*](?|!)[0..1] : (dss[1..*] : Dynamic|System,
8                ass[0..1] : Assignment, dsd[1..*] : Dynamic|System)
9                { Condition{}; Discrete{}; };
10   };}
```

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 '$A \mid\mid B$'. 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. In Apricot, 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.

```
1  interface Controller{
2      Constraint clock;
3      Requires dy[1..*] : Dynamic;
4      Requires ass[1..*] : Assignment;
5      void Composition(){
6        Requires coms[1..*](?|!)[0..1] : (dys[1..*] : Dynamic,
7                 ass[0..1] : Assignment, dyd[1..*] : Dynamic)
8                 { Condition{}; Discrete{}; };
9      };}
```

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.

```
1  interface Dynamic{
2      void Continuous();
3      Invariant{};
4      Native void Start();
5  }
```

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.

```
1  interface Assignment{
2      void Discrete();
3  }
```

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.

C. Polymorphism

Polymorphism is a principle in biology in which an organism or species can have various forms or stages. This principle can also be applied on hybrid systems modeling as in the Apricot language. Subclasses can define specific behaviors while share some of the same functionality of the parent class.

Here, we employ an example of polymorphism in our running case. The class $DoorSystem$ have two subclass $ScreenDoorSystem$ and $TrainDoorSystem$ for platform screen doors and train doors, respectively. The methods $getDoorAmount$ and $registerComposition$ are shared by these two subclass. But, in each subclass, the `Init` methods declares different behaviors.

```
1 class DoorSystem implements System{
2   ...
3   int getDoorAmount(){ ... }
4
5   void registerComposition
6   (Door[] doors, DoorController controller){ ... }
7
8   void Init(){ ... }
9 }
```

The `Init` method for screen doors system has its own unique behaviors,

```
1 class ScreenDoorSystem extends DoorSystem{
2   ...
3   void Init(){
4       registerComposition(PADoors,PADController);
5       registerComposition(PBDoors,PBDController);
6
7       //start dynamic closed of doors on Platform_A
8       PADoors.closed.Start();
9       //start dynamic closed of doors on Platform_B
10      PBDoors.closed.Start();
11
12      //start controller of Platform_A
13      PADController.closed.Start();
14      //start controller of Platform_B
15      PBDController.closed.Start();
16  }
17 }
```

Meanwhile, the `Init` method for train doors system also has specific behaviors that are different from the class $ScreenDoorSystem$.

```
1 class TrainDoorSystem extends DoorSystem{
2   ...
3   void Init(){
4       registerComposition(doors,TDController);
5       //start dynamic closed
6       doors.closed.Start();
7
8       //start controller for train doors
9       TDController.closed.Start();
10  }
11 }
```

In this case, the two classes $ScreenDoorSystem$ and $TrainDoorSystem$ have the same type of $DoorSystem$, but they can behave differently when the method `Init` is called.

V. Case Study: Subway Control System

As described in Section III, in our case, we consider the distance between trains in a subway control system. During the running, the train shall brake when its distance to the train ahead is less than or equal to 300 meters (urgent distance).

Fig. 7 illustrates the hybrid automaton for urgent control in UPPAAL [17]. The tool supports arrays, user-defined methods, and control structures, but it does not allow continuous variables in user-defined methods. To implement the control logic of urgent distance control in an hybrid automaton is not a simple process, and it is hard to maintain the control logic in the automaton. Even we only have four modes in the automaton, the relations between control switches and arrows are difficult to be distinguished.

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. 8(a), the class *Calculating* is used by the controller. The distance is calculated in the dual-for-statements[1] (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 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.

For the modeling tool *XtextApricot*, we illustrate the view of class *DistanceController* in Fig. 8(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 I.

VI. Operational Semantics

Usually, structural operational semantics ([18], 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.

A. 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

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

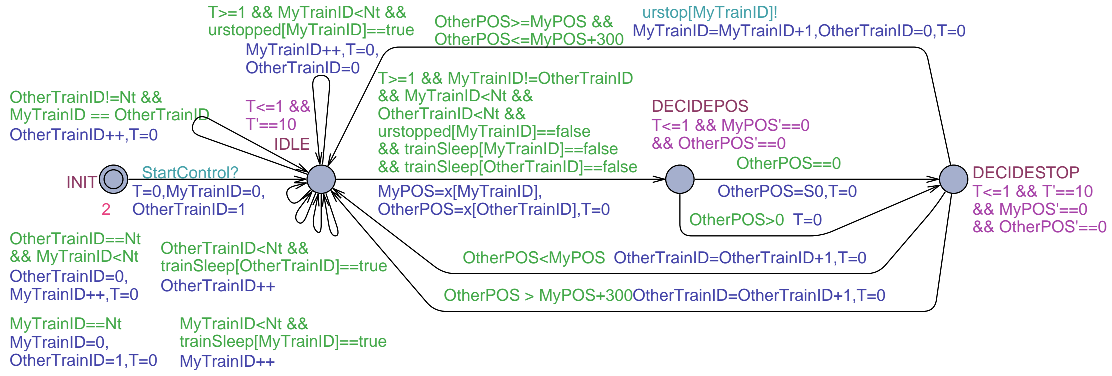
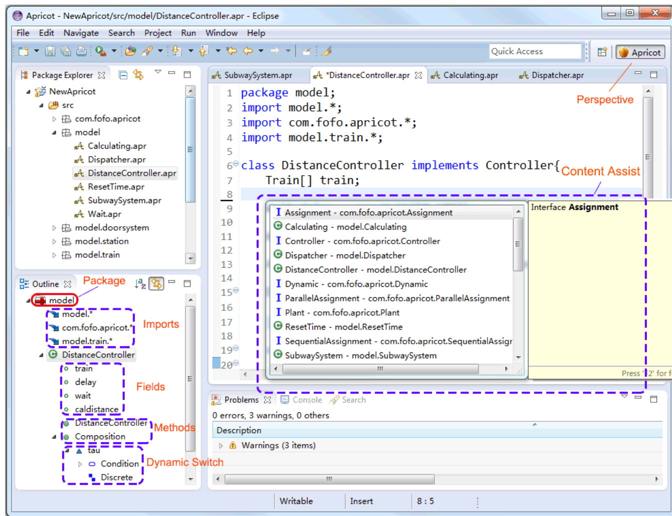


Fig. 7. The hybrid automaton for urgent control in UPPAAL. In the model, 'StartControl' and 'urstop[MyTrainID]' are channels (i.e., synchronization labels in hybrid automata). Channels are used to synchronize processes (i.e.,hybrid automata). This is done by annotating edges in the model with synchronisation labels. They are of the form $e?$ or $e!$, where e is a side effect free expression evaluating to a channel, thus, 'urstop' is an array of channels. The continuous behavior in the model is specified as the form of $v' == e$, for instance, 'T'==10' denotes the differential equation $\frac{dT}{dt} = 10$

```
 1 package model;
 2 import com.fofo.apricot.SequentialAssignment;
 3 import model.train.Train;
 4 class Calculating implements SequentialAssignment{
 5  Train[] train;  //the array of trains
 6  Calculating(Train[] train){ //constructed by an array of trains dynamically
 7    this.train = train;
 8  }
 9  void Discrete(){
10    int mindis = Inf;
11    for(Train currTrain : train){
12      int currdir = currTrain.getCurrentDirection();
13      real currpos = currTrain.getCurrentPosition();
14      for(Train otherTrain : train){
15        int otherdir = otherTrain.getCurrentDirection();
16        real otherpos = otherTrain.getCurrentPosition();
17        if(currTrain!=otherTrain and currdir == otherdir){
18          //when two trains are different, but in same direction
19          //calculate the distance between them
20          real distance = currdir * (otherpos - currpos);
21          if(distance<=300 and distance>=0 and distance < mindis){
22            mindis = distance;
23          }
24        }//end calculate of distance
25      }
26      if(mindis < Inf){//the initial value of mindis is Inf
27        currTrain.urStop(!);//stop currTrain
28        mindis = Inf;
29      }
30      else if(currTrain in currTrain.urgent_stop){//currTrain can be restart
31        currTrain.urStart(!);
32      }
33    }//end for-loop
34  }//end discrete()
35 }//end class Calculating
```

(a) The calculation for distance between trains



(b) The modeling tool

Fig. 8. Model view of Apricot

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 3 (Configurations): We define the set of configurations with statements, states, and types, formally as follows:

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

where $1 \le i \le n$, 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.

B. 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 $x \in$ **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:

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

$$\frac{\langle \mathcal{P}re.S_1, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma'_1, \tau \rangle}{\langle \mathcal{P}re.(S_1; S_2), \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.S_2, \sigma'_1, \tau \rangle} \qquad \text{[sequential-assignment]}$$

$$\frac{\langle \mathcal{P}re.S_1, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma'_1, \tau \rangle, \quad \langle \mathcal{P}re.S_2, \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma'_2, \tau \rangle}{\langle \mathcal{P}re.(S_1 || S_2), \sigma, \tau \rangle \rightarrow \langle \mathcal{P}re.Skip, \sigma'_3, \tau \rangle} \qquad \text{[parallel-assignment]}$$

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.

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

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

(i) Differential Equation D_e of Continuous Variable v. Suppose that there exists a function $f : I \rightarrow \mathbb{R}$, 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, $f : I \rightarrow \mathbb{R}$, $I = [a, b]$, $f(t)$ 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,

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

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

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,

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

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,

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

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,

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

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,

$$\frac{\langle \mathcal{P}re.(CR_s \; and \; CR_t), \sigma, \tau\rangle \to True, \quad \langle \mathcal{P}re.(A_s || A_t), \sigma, \tau\rangle \to \langle \mathcal{P}re.Skip, \sigma', \tau\rangle, \quad \langle \mathcal{P}re.(D_{s_2}.Inv \; and \; D_{t_2}.Inv), \sigma', \tau\rangle \to True}{\langle \mathcal{P}re.(CR_s || CR_t), \sigma, \tau\rangle \to True} \text{[valid-sync]}$$

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

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

(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 $CR_s \sim CR_t$ is defined as follows:

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

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 $CR_s \sim CR_t$ is defined as

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

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

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

$$\frac{\langle \mathcal{P}re.R_s(!)(D_{s_1}, A_s, D_{s_2}, C_s), \sigma, \tau\rangle \to True, \quad \langle \mathcal{P}re.R_s(?)(D_{s_1}, A_s, D_{s_2}, C_s), \sigma, \tau\rangle \to True}{\langle \mathcal{P}re.(CR_S \sim CR_t), \sigma, \tau\rangle \to \langle \mathcal{P}re.(CR_S || CR_t), \sigma, \tau\rangle} \text{[cond-sync]}$$

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.

```
1 void Init(){
2       Initialize(Variables);      // task-1
3       Register(Compositions);     // task-2
4       Start(Dynamics);            // task-3
5 }
```

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:

```
1 void Composition(){
2     CR(D1,A,D2){C};
3 }
```

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.

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

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.

VII. Conclusion and Future Work

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

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 behaviour, or append new behaviour, making the system more and more complicated. The third is the composition, it constructs the relationship between objects, components and subsystems, making the system model scalable and distinct for implementation.

In the future, an important planed feature 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 [19]. For the feature of dynamic object instantiation, it can be reconfigured into a special form of uncertainty of system behaviour, 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 state-charts in Matlab Simulink/Stateflow. This translation can be conveniently implemented within the Xtext framework.

References

[1] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho, "Hybrid automata: An algorithmic approach to the specification and analysis of hybrid systems," in *Hybrid Systems*, ser. LNCS. Springer-Verlag, 1993, vol. 736, pp. 209–229.

[2] J. He, "From csp to hybrid systems," in *A Classical Mind, Essays in Honour of C.A.R. Hoare*. Prentice Hall International, 1994, pp. 171–189.

[3] C. Zhou, J. Wang, and A. P. Ravn., "A formal description of hybrid systems," in *Hybrid Systems III*, ser. LNCS. Springer-Verlag, 1996, vol. 1066, pp. 511–530.

[4] P. J. L. Cuijpers and M. A. Reniers, "Hybrid process algebra," *The Journal of Logic and Algebraic Programming*, vol. 62, no. 2, pp. 191–245, 2005.

[5] A. Platzer, *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer-Verlag, 2010.

[6] T. Henzinger, P. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 110–122, 1997.

[7] E. Asarin, T. Dang, and O. Maler, "The d/dt tool for verification of hybrid systems," in *Proceedings of CAV'02*, ser. LNCS. Springer-Verlag, 2002, vol. 2404, pp. 365–370.

[8] G. Frehse, "Phaver: algorithmic verification of hybrid systems past hytech," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 3, pp. 263–279, 2008.

[9] L. Zhang, Z. She, S. Ratschan, H. Hermanns, and E. Hahn, "Safety verification for probabilistic hybrid systems," in *Proceedings of CAV'10*, ser. LNCS. Springer-Verlag, 2010, vol. 6174, pp. 196–211.

[10] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proceedings of CAV'11*, ser. LNCS, vol. 6806. Springer-Verlag, 2011, pp. 379–395.

[11] A. Platzer and J.-D. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems." in *IJCAR*, ser. LNCS, A. Armando, P. Baumgartner, and G. Dowek, Eds., vol. 5195. Springer-Verlag, 2008, pp. 171–178.

[12] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[13] M. Fowler, *Domain-specific languages*. Addison-Wesley Professional, 2010.

[14] F. D. Torrisi and A. Bemporad, "Hysdel-a tool for generating computational hybrid models for analysis and synthesis problems," *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pp. 235–249, 2004.

[15] P. Fritzson and V. Engelson, "Modelica – a unified object-oriented language for system modeling and simulation," in *Proceedings of ECOOP'98*, ser. LNCS. Springer-Verlag, 1998, vol. 1445, pp. 67–90.

[16] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, P. Mishra, G. Pappas, and O. Sokolsky, "Hierarchical modeling and analysis of embedded systems," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 11–28, 2003.

[17] G. Behrmann, A. David, K. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "UPPAAL 4.0," in *Proceedings of QEST*, 2006, pp. 125–126.

[18] G. D. Plotkin, "A structural approach to operational semantics," Department of Computer Science, Aarhus university, Tech. Rep. DAIMI FN-19, September 1981.

[19] P. Cousot, R. Cousot, and L. Mauborgne, "Theories, solvers and static analysis by abstract interpretation," *Journal of the ACM*, vol. 59, no. 6, pp. 31:1–31:56, 2012.