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. We propose Apricot as an object-oriented language for modeling hybrid systems. The language combines the features in domain specific language and object-oriented language, that fills the gap between design and implementation, as a result, we put forward the modeling language with simple and distinct syntax, structure and semantics. In addition, we introduce the concept of design by convention into Apricot. As the characteristic of object-oriented and the component architecture in Apricot, we conclude that it is competent for modeling hybrid systems without losing scalability.

Keywords: Apricot, Object-Oriented Modeling, Hybrid Systems, Design by Convention

1 Introduction

Hybrid systems are concerned about the discrete control mode transitions, the continuous physical behavior, and the interaction between these two parts. As mentioned in [7], the design of a system is the process that building a concrete to carry out some goals. Meanwhile, people in the hybrid systems domain have the ambition to control their environment, i.e., the physical world. For hybrid systems, numerous modeling approaches had been proposed, the hybrid automata [1,2], Hybrid CSP [13,21], HyPA (hybrid process algebra) [6], and hybrid program [16], etc. Regarding the formal verification on hybrid systems, various tools can be used, for instance, HyTech [15], d/dt [5], PHAVer [9], SpaceEx [10], and KeYmaera [17]. These works are respectable and formal, the common feature is that most of them are focus on the high level abstraction of hybrid systems. However, industrial applications of formal methods need a great level of abstraction in existing development processes and an easier manner to adopt for users. In other words, usability and complexity hiding are the major concerns for designers and developers in industry. Modelica [11] is a multi-domain object-oriented modeling language, it involves systems relating electrical, mechanical, control, and thermal components, etc. And, one of the characteristics of Modelica is that, the class in Modelica can not be executed explicitly, but simulated by a simulation engine. From the 1.0 release in 1997 when it began to model continuous

dynamic systems to the 3.3 release in May, 2012 the addition of periodic and non-periodic synchronous controllers, the revision of Modelica has never been ceased. The description capability of Modelica is powerful, and the applications of Modelica is pervasive. Nevertheless, it is not designed for formal verification, although it is quite suitable for simulation. The reason is that the semantics of Modelica is prone to be deterministic, however in the area of hybrid systems, it is prone to consider the non-deterministic evolution of the system behavior.

The motivation to propose the language Apricot is that, we want to construct an object-oriented language for modeling hybrid systems. The language should satisfy the following requirements. First, clear and simple syntax. We know that binary code is accurate and precise, so why people in the highly developed modern society do not use binary code as the communication language in life. Because binary code is closed to hardware, it is far from daily life and hard to be acquainted. The same is in the area of hybrid systems. A language that is close to the designers and developers in industry is needed and worthy to be developed. Second, distinct structure. As an object-oriented language, we can employ design patterns [12] in the system design process. For instance, to demonstrate the hierarchical structure of complex hybrid systems, we utilize the composition pattern to build the ownership relation between global system and subsystems. Composition pattern in Apricot constructs the tree structure with respect to objects of *System*, *Plant*, *Dynamic* and the subsystems of *Plant* object. We treat objects of *Dynamic* and *System* as a similar way under the compositional relationship, it results in the ownership between plant and subsystem, and then the relationship between system and subsystem. The third is an explicit semantics. We propose the operational semantics for Apricot. As the highly structural style of Apricot models, the semantics is clear and compositional.

The contributions of our work can be elaborated as follows. The first is about the innovation on the Interface conception. Interface is an abstraction of the type, a suitable Interface for hybrid systems should consider the relations for system components and in favor of the hierarchical structure construction for complex systems. The common constraints and conventions are better to be defined in the abstract level than in the implementation part. Because, the higher the common knowledge is the easier the developer to know well. Traditionally, in object-oriented languages, the Interface only contains methods and no instance variable declaration or just the constant (in Java, or property in C#, etc.) is allowed. In Apricot, we allow variable requirements, constraint indications and built-in block statements in the Interface. The variable requirements define the relationships between the current type and other types. Therefore, it has the ability to describe the ownership among different components. The constraint indications denotes the behavior that is forced to conform. For instance, the *clock* constraint indication for the *Controller* Interface set the derivative of the variable of *Controller* to be the constant number one. The built-in block statement denotes the right usage and position that the block should be. In Apricot, for example, the *Condition* block is positioned in the *Composition* method of the Interface *Plant*. As a consequence, the innovation enhances and clarifies the relationship for various system components by variable requirements, specifies the

limitation of some components by constraint indications, and explicitly states the proper usages of blocks by the built-in block statement declaration in Interface.

Moreover, we apply the principle of *Architecture as Language*, and build the combination of the features from Domain Specific Language (abbreviated as DSL, [20,8]) and Object-Oriented Language (abbreviated as OOL). The DSL notations (such as the variable requirements and constraint indications) used in Apricot are good for the building of component architecture, and as a result, it makes easier to communicate with domain experts during the system design process. On the other hand, the OOL is familiar to developers in industry, and close to the implementations of the system. The combination of DSL and OOL in Apricot fill the gap between the design at higher level and the implementation for the concrete. This paper is organized as follows. Section 2 describes the syntax of Apricot and an example (bouncing ball) modeling under Apricot. The operational semantics is demonstrated in Section 3. In addition, Section 4 discusses the features of design by convention in Apricot. And, we make the conclusions in Section 5.

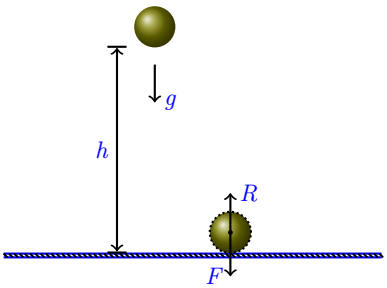
2 Syntax of Apricot

In this section we will describe the basic syntax of Apricot. As a modeling language for hybrid systems, one has to consider the hierarchical structures of the system to demonstrate the modularity features, and also has to propose the definitions of system dynamics with the relations between continuous flow and discrete assignments. The following recursive definitions have cover the overview of the above ambition.

$$
\begin{aligned}
System &::= ParaPlants \parallel ParaContrs;\\
ParaPlants &::= \|_{i=1}^{n}\ Plant_i;\\
ParaContrs &::= \|_{i=1}^{m}\ Controller_i;\\
Plant &::= AtomicComp \mid Comp(Dynamic^{+}, Assignment^{+}, System);\\
Controller &::= AtomicComp;\\
AtomicComp &::= Comp(Dynamic^{+}, Assignment^{+});\\
Assignment &::= SequentialAssignment \mid ParallelAssignment.
\end{aligned}
$$

where $n, m \in \mathbb{Z}^{+}$(positive integers), symbol '$\|$' denotes parallel composition. '$Dynamic^{+}$' represents a set of *Dynamic* objects, and '$Assignment^{+}$' has the similar meanings (*Assignment* objects).

The system defined here has the point that each system contains one or more plants and controllers. This is different from other approaches or languages such as hybrid automata which do not have this restrict.



(a) The ball. h denotes the height of the ball, g is for the acceleration of gravity.

```
Dynamic Moving{
 Real height,velocity,acceleration;
 /*Constructor*/
 Moving(Real height,Real velocity,Real acceleration){
       this.height=height;
       this.velocity=velocity;
       this.acceleration=acceleration;
 }
 Continuous(){
       dot(height,1) == velocity;
       dot(velocity,1) == -acceleration;
  }
 Invariant{
       height in [0,15];
       velocity in [-60,60];
 };
}
```

(b) Class Moving implements interface Dynamic.

```
ParallelAssignment Jump{
 Real height,velocity,coefficient;
 /*Constructor*/
 Jump(Real height,Real velocity,Real coefficient){
       this.velocity = velocity;
       this.height = height;
       this.coefficient = coefficient;
 }
 Discrete(){
       velocity = -coefficient * velocity;
       height = height;
 }
}
```

(c) Class Jump implements interface ParallelAssignment.

```
Plant Ball{
 Real height,velocity,k,g;
 Ball(Real height, Real velocity, Real k, Real g){
     this.height = height;
     this.velocity = velocity;
     this.k = k;
     this.g = g;
 }
 Dynamic moving = new Moving(height,velocity,g);
 Assignment jump = new Jump(velocity,height,k);
 Composition(){
     CompMJ(moving,jump,moving){
         Condition{  moving.height==0; };
     };
 }
}
```

(d) Class Ball implements interface Plant.

```
Controller God{
 Real mass,height,velocity,k,t,g;
 God(Real mass, Real height, Real velocity,
 Real k, Real t, Real g){
     this.mass = mass;
     this.height = height;
     this.velocity = velocity;
     this.k = k;
     this.t = t;
     this.g = g;
 }
 Dynamic idle = new Dynamic(){
     Continuous(){ dot(t,1)==1; }
 };
 Assignment reset = Skip;

 Composition(){
     CompIR(idle,reset,idle){
       Condition{
        height == 0;
        Resiliency(mass,velocity,k)>mass*g;};
     };
 }
}
```

(e) The controller of bouncing ball system, class God implements interface Controller.

```
System BouncingBall{
 Real height,velocity,t;
 Real h[] = {15, 10, 12};
 Real v[] = {0, 1, 1.5};
 Constant real g=9.8,k=0.6,mass=5;
 Controller god=new God(mass,height,velocity,k,t,g);
 Plant ball = new Ball(height,velocity,k,g);
 BouncingBall(){
       god.CompIR || ball.CompMJ;
       god || ball; //maybe not necessary
 }
 Init(){
       height=h[1],velocity=v[1],t=0;
       god.idle.start();
       ball.moving.start();
 }
}
```

(f) Class BouncingBall implements interface System.

Fig. 1. Bouncing ball model.

(C.1) *Continuous*(){
$dot(Var_1, Nat_1) == MathExp_1;$
$dot(Var_2, Nat_2) == MathExp_2;$
$\cdots$
$dot(Var_n, Nat_n) == MathExp_n;$
}

(C.2) *Invariant*{
$Variable_1$ *in* $\lfloor Real_1, Real'_1 \rceil;$
$Variable_2$ *in* $\lfloor Real_2, Real'_2 \rceil;$
$\cdots$
$Variable_n$ *in* $\lfloor Real_n, Real'_n \rceil;$
};

(C.3) *Discrete*(){
$Variable_1 = MathExp_1;$
$Variable_2 = MathExp_2;$
$\cdots$
$Variable_n = MathExp_n;$
}

(C.4) *Condition*{
$MathExp_1$ *Rel* $MathExp'_1;$
$MathExp_2$ *Rel* $MathExp'_2;$
$\cdots$
$MathExp_n$ *Rel* $MathExp'_n;$
};

Dynamic object is an instance of the class that implements the *Dynamic* interface. *Dynamic* object is refers to flows which are used to model continuous behavior of physical plants. The implementation class of *Dynamic* interface defines the continuous valuations of the variables in the system over time. And, it also specifies the invariant of the continuous flow. The *Continuous* method in the *Dynamic* implementation class has the form as depicted in (C.1), in which, for $1 \leq i \leq n, Var_i$ is the variable of the system, natural number Nat_i represents the derivative order of Var_i that is not equal to 0, $MathExp_i$ 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 order variables, 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* defines the mathematical function defined by the designer or the built-in function in Apricot. Such as addition, subtraction, multiplication, division, etc. For example, the multiplication in Fig. 1(c) is an infix form function.

The *Invariant* statement specifies the properties of the system during the continuous evolution, as illustrated in (C.2). In which, *Real* denotes the real number, $\lfloor \in \{ \, '(', '[' \, \}$, and $\rceil \in \{ \, ')', ']' \, \}$. Symbols $'(', \, ')'$ are used to define open intervals, and $'[', \, ']'$ for closed intervals. For example, in Fig. 1(b), '`height in [0, 15]`' clarifies the variable `height` evaluates the value within the closed interval [0, 15] during the continuous evolution. Note that, the left-open parenthesis is limited to the special real number `-Inf`, and the right-open parenthesis is limited to `Inf`, thus intervals like $(1, 2)$, $(-\texttt{Inf}, \texttt{Inf}]$, $[-\texttt{Inf}, \texttt{Inf})$ and $[-\texttt{Inf}, \texttt{Inf}]$ is invalid.

Assignment interface has two sub-interfaces, *SequentialAssignment* and *ParallelAssignment*. Both implementations have a discrete method with the form in (C.3). If this discrete method is defied in class implementing the interface *SequentialAssignment*, then it is the sequential composition of these n assignment statements. Otherwise, if it is defined in class implementing the interface *ParallelAssignment*, then the parallel composition is the semantics that the assignment statements are supposed to represent. Fig. 1(c) is an example of *ParallelAssignment* implementation.

The *Composition* statement connects the *Dynamic* object and *Assignment* object by a *Condition* statement. The *Condition* statement has the form in (C.4). In which, $Rel \in \{==, <, >, <=, >=, !=\}$ is the relation operator, and the expression "$MathExp_i\ Rel\ MathExp'_i$;" defines the relation between the evaluations of $MathExp_i$ and $MathExp'_i$. For example, in Fig. 1(d), the *Composition* method refers to *Dynamic* object `moving` and *Assignment* object `jump` with `moving.height==0`. Therefore, if the value of the variable `height` in `moving` is equal to 0 (i.e., the ball hits the ground), then the *Assignment* `jump` will be executed and the control will move on to `moving` after this execution provided that the invariant is satisfied.

Example 1. Bouncing ball is a traditional model in hybrid system. The system has a controller named `god` and a plant named `ball`. The controller has *Dynamic* `idle`, *Assignment* `reset` and the *Composition* relation `CompIR` paralleled with plant's `CompMJ`. The plant has *Dynamic* `moving`, *Assignment* `jump` and the *Composition* `CompMJ` paralleled with controller's `CompIR`. The two source-free arrows in the plant `ball` and controller `god` represent the initial dynamics. Therefore, `moving` and `idle` are the initial dynamics of `ball` and `god`, respectively.

Fig. 1(a)–1(f) are the model code for the bouncing ball system. Fig. 1(a) depicts the ball, when the ball hits the flat horizontal ground, it suffers the gravity F and the elastic force R. The class `Moving` (in Fig.1(b)) is an implementation of the `Dynamic` interface. It declares that the first order derivative of `height` over time equals `velocity`, and the first order derivative of `velocity` over time is equal to `-acceleration`. In Fig. 1(d), an object named `moving` is created with the type of class `Moving`, and relates the variables `height`, `velocity`, `g` of class `Ball` to `height`, `velocity`, `acceleration` in class `Moving`, respectively.

2.1 Class, Object and Relation

Class declaration defined reference types. The body of class declaration defines the implementation details. All classes are non-nested in Apricot. This means that the class declaration defined within the body of another class or interface is invalid.

The body of a class consists of fields, methods, instance, relations, and constructors. Field declarations describe instance variables, each instance of the class holds a new substantiation of the instance variable.

Class Declaration. We have three kinds of class declaration:

– Top-level Class. If the class do not have super class, and do not implements any other interface:

$$\begin{array}{l} Class\ Identifier\{ \\ \qquad ClassBody \\ \} \end{array}$$

in which, we do not specify the access modifiers (e.g. *Public, Protected, Private* in Java). The keyword `this` in the constructor denotes the current instance being constructed. If keyword `this` occurs in an instance method then it represents the object for which the method was defined. Most of the time, the keyword `this`

is employed to distinguish the instance variable from parameter variables when the names of variables in different classes clashed.

– Interface Implementation. If one class implements an interface, the class declaration is:

$$\begin{array}{l} InterfaceType\ Identifier\{ \\ \quad ClassBody \\ \} \end{array}$$

It is difference from many other object-oriented languages (e.g., Java, C++), we do not use the keyword *implements* to specify the interface type the class implements here. In example 1, the classes (see Fig.1(b)–1(f)) are all interface implementations.

– Inheritance. If one class extends other class (i.e. SuperClass), the class declaration:

$$\begin{array}{l} ClassType\ Identifier\{ \\ \quad ClassBody \\ \} \end{array}$$

Constructor Declaration. The constructor takes the responsibility for the creation of an instance of a class. Moreover, it weaves the connection between different components in Apricot models. The constructor declaration as follows for the case that formal parameters are presented:

$$\begin{array}{l} Identifier(Formal\ Parameters)\{ \\ \quad ConstructorBody \\ \} \end{array}$$

For example, in Fig. 1(d), the `Ball(...)` constructor is:

```
Ball(Real height, Real velocity, Real k, Real g){
    this.height = height;
    this.velocity = velocity;
    this.k = k;
    this.g = g;
}
```

The formal parameters are a list of parameter specifiers and separated by the comma symbol ‘,’. Each parameter specifier is a pair of a type and an identifier. The identifier is the name of the parameter. In Fig.1(f) line 7, it creates a `Ball` object using the ‘`Ball(...)`’ constructor. Meanwhile, it creates the connection of variables (`height, velocity, k, g`) in *system* `BouncingBall` with the variables (`height, velocity, k, g`) in *plant* `Ball`. The statements in the constructor of `Ball`, e.g. “`this.height = height`” makes the instance variable `height` of `Ball` and the instance variable `height` of `BouncingBall` refer to the same entity. All the modification on variable `height` take place in `Ball` or `BouncingBall` will be recognized immediately by each other.

Formal parameters can be absent, for the case of line 8 in Fig.1(f). The line 9 of the constructor denotes that the composition relation `CompIR` of controller `god` is parallel with the composition reltion `CompMJ` of plant `CompMJ`. The line 10 denotes that the controller `god` is parallel with the plant `ball`. The initializer is declared by the method “`Init`(){...}” at line 12 $\sim$ 16.

Initializer Declaration. The initializer method specifies the initial values of the instance variables in a system. For example, the line 13 in Fig.1(f) sets the initial value of `height` to the number 15, `velocity` the number 0 and the initial value of `t` the number 0. In addition, it starts the initial dynamics of the components in the system. For instance, the initial dynamic of *controller* `god` is `idle` and the initial dynamic of *plant* `ball` is `moving` specified by line 14 and line 15 in Fig.1(f), respectively.

Anonymous Class Declaration. Anonymous class is an implementation of an interface or an inheritance of a super class. In Fig. 1(e), the variable `idle` declared at line 12 refers to an instance of an anonymous class which implements the interface `Dynamic`. The method `Continuous` defined in the anonymous class denotes the first order time-derivative of variable `t` is equal to 1. Therefore, variable `t` takes the role of a clock.

Moreover, no invariant is defied in the anonymous class, which means that it has an implicit invariant `Ture`, variable `t` can take any value in real numbers $\mathcal{R}$. Anyway, as time is not negative, we can specify an invariant that `t` is always equal to or greater than the number 0:

```
Invariant{ t in [0,Inf); };
```

where, '`Inf`' denotes the infinity $+\infty$.

2.2 Interface, Inheritance and Relationship

In Apricot, there are five built-in interfaces, each defines one key element of the Apricot model. The built-in interface may consist of four parts: method signatures, variable requirements, constraint indications and built-in block statements. From now on, these four parts are abbreviated to MVCB in this paper. Method signature defines the name and arguments of the method. Variable requirement holds the relations between the current interface and other interfaces, it also restrict the count of objects of the proper types. Constraint indication demonstrates the limitation for the behavior of the object which implements the interface. And, the built-in block statement positioned in the interface emphasizes the structure of the language, and indicates the right place for the application of the special statement.

– **System Interface** depicted in (I.1), where, '*Requires*' is a keyword in Apricot, '1..∗' denotes at least one entity. Therefore, each *System* object contains one or more than one *Plant* object, and it also for the objects of type *Controller*. The method signature '$Init()$' indicates that the *System* has an initializer that do not contain any argument and no return value for this initializer. '*plants*' and '*controllers*' are the names of the variables referring to the proper types behind the colon symbol (':').

– **Plant Interface** depicted in (I.2), where, it indicates that the implementation of this interface holds several objects of the type *Dynamic* and *Assignment*, and may have a subsystem or not. The *Composition* method is used for defining the composition relationships between *Dynamic* (or *System*) objects and *Assignment* objects. Each composition relationship with respect to three arguments:

the source, action, and the destination. And, the form '$(dysy[.], ass[.], dysy[.])$' in the composition relationship shows that $dysy[.]$ is the source (*Dynamic or System*), ass[.] is the action, and $dysy[.]$ (also can be *Dynamic or System*) is the destination, '.' represents the proper index. The composition relationship denotes the control switch that from the source to the destination under the conditions defied in the *Condition* block statement. During the control switch the action which is restricted to the *Assignment* object (i.e., '$ass[.]$') is executed.

– **Controller Interface** depicted in (I.3), where, it is the same as *Plant* except the *Constraint Indication* and the absent of subsystem. The *clock Constraint Indication* '*Constraint clock*' denotes that the differential equations in the *Dynamic* object of *Controller* have the restriction: the derivative assigned to the variable is restrict to number 1.

– **Dynamic Interface** depicted in (I.4), where, it indicates that each *Dynamic* implementation has a method and an built-in *Invariant* block statement. The method '*Continuous*()' with respect to the continuous evolution of the system states. The form of the method has been declared before in Sect 2. The *Invariant* is applied to define the range of proper variable concerned for the current *Dynamic* object.

– **Assignment Interface** depicted in (I.5), the *Assignment* interface only has the method '*Discrete*()'. The *Discrete* method plays the role of the actions that would be executed during the control switch of dynamics. Moreover, there are two interfaces inherit the *Assignment* interface, *SequentialAssignment* and *ParallelAssignment*. *SequentialAssignment* has the semantics of sequential composition for its assignment statements, and *ParallelAssignment* has a parallel composition semantics.

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

(I.2)
```
Interface Plant{
  Requires dy[1..*] : Dynamic;
  Requires ass[1..*] : Assignment;
  Requires sy[0..1] : System;
  Composition(){
   Requires coms[1..*] : (dysy[.], ass[.], dysy[.])
         { Condition{}; };
  };
}
```

(I.3)
```
Interface Controller{
  Constraint clock;
  Requires dy[1..*] : Dynamic;
  Requires ass[1..*] : Assignment;
  Composition(){
   Requires coms[1..*] : (dy[.], ass[.], dy[.])
         { Condition{}; };
  };}
```

(I.4)
```
Interface Dynamic{
  Continuous();
  Invariant{};
}
```

(I.5)
```
Interface Assignment{
  Discrete();
}
```

In addition, as the existence of MVCB in the interface declaration, 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. And, the implementation

of interface in Apricot should consider to implement and follow the MVCB in the implemented interface.

3 Operational Semantics

Structural operational semantics ([18,19], SOS) was proposed by G.D.Plotkin in 1981. Transition system is the base for structural operational semantics. It takes the transition relation between configurations to characterize the operational feature of system behaviour. Usually, 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. For the semantics and verification of object-oriented languages, some related works can be found in [3,4,14].

Definition 1. *A Transition System (***TS***) is a structure consists of a set of configurations (***C***) and the relation (*$\rightarrow$*) between configurations, i.e.,* $\mathbf{TS} \stackrel{def}{=} \langle \mathbf{C}, \rightarrow \rangle$*, where* $\rightarrow \subseteq \mathbf{C} \times \mathbf{C}$*.*

3.1 Configurations

Any insight into a hybrid system is obtained through the state of the system. Each state is a valuation of the variables in the system. After the system start-up, it always accompanied with a state at each time point. All the states compose a state space of the system. Based on the state space, one can check whether some specific state can be reached by the system for some proper initial states. It is called the reachability analysis. And, various respectable works had been done, e.g., the Hytech [15] proposed by Henzinger etc., the Phaver [9] and SpaceEx [10] by Frehse etc., the hybrid process algebra approach [6] by P.J.L. Cuijpers, and Platzer's dynamic differential logic [16], etc.

Besides system states, to reveal the relation between statement and state, we also need to pay attention to the statements (control flow) throughout the system execution. These understanding can be used to check the statement-related properties. For example, we can check that some particular dynamic method is not reached or executed by the system with the knowledge of both statement and state.

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

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

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 consists of all functions that mapping from the set of variables* **Vars** *to the set of values* **Vals***,* **T** *is a set of functions which relate each variable in* **Vars** *with a type in* **Types***.*

A prefix annotated statement is a linked list that begins with a variable (ϑ_1) which denotes the system and ended with the statement (ϑ_n) currently executed or expression to be evaluated. Along the list there will be objects or methods. An Apricot model comprises more than one component, and these components paralleled. As a result, the first element of a configuration is a subset of Θ, consists of the parallel prefix annotated statements. (Fig. 2 illustrates the example prefix annotated statements for bouncing ball system)

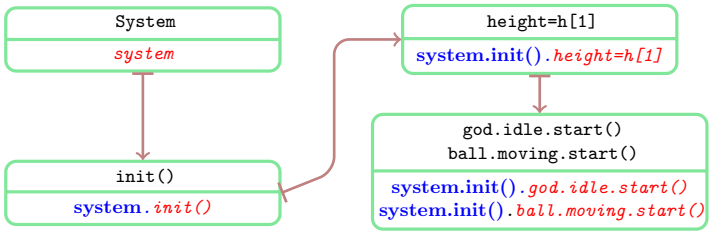


Fig. 2. The example of prefix annotated statements for bouncing ball system. The italic statement is the current statement the system executed.

Moreover, considering the nondeterminism feature of Apricot, a model of Apricot consists of numerous prefix annotated statements, thus all the possible runs of the model can be illustrated by a tree structure, and each branch may has a different state space.

3.2 Axioms and Rules

Here, we will give the axioms for Apricot. Consider single statement θ, for $\{\mathbf{Pre}.\theta\} \in \mathcal{P}(\Theta)$, $\sigma \in \mathcal{P}(\Sigma)$, and $\tau \in \mathcal{P}(\mathbf{T})$, then $\langle\{\mathbf{Pre}.\theta\}, \sigma, \tau\rangle \in \mathbf{C}$. For simplicity, we take $\mathbf{Pre}.\theta$ for $\{\mathbf{Pre}.\theta\}$ in the following axioms ($\mathbf{Pre}$ is the prefix):

– **Arithmetic expression** e.

Evaluation of constant numbers:

$$\langle \mathbf{Pre}.n, \sigma, \tau\rangle \to n, \tag{1}$$

where n is a constant number.

Evaluation of variable:

$$\langle \mathbf{Pre}.v, \sigma, \tau\rangle \to n, \tag{2}$$

where, v is a variable of number type, and $\sigma(v) = n$.

Evaluation of addition:

$$\frac{\langle \mathbf{Pre}.e_1, \sigma, \tau\rangle \to n_1 \quad \langle \mathbf{Pre}.e_2, \sigma, \tau\rangle \to n_2}{\langle \mathbf{Pre}.(e_1 + e_2), \sigma, \tau\rangle \to n}, \tag{3}$$

where, e_1 and e_2 are variables or constant numbers, and n is the summation of n_1 and n_2.

– **Mathematical function expression**.

Derivative over time t with order n:

$$\langle \mathbf{Pre}.dot(v, n), \sigma, \tau\rangle \to \frac{d^n v}{dt^n}, \tag{4}$$

where, $\frac{d^n v}{dt^n}$ is a formula that represents the n-th order derivative of v over time. In fact, we can regard the n-th order derivative as an attribute or observation of the variable, and employ a new variable to maintain the value of the derivative. We produce a new variable when it occurs at the first time, and the name would be v_n. Thus, (4) is changed to

$$\frac{\langle v\_n, * \rangle \notin \sigma}{\langle \mathbf{Pre}.dot(v, n), \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.v\_n, \sigma', \tau' \rangle}, \tag{5}$$

where, symbol '*' stands for any value, $\sigma' = \sigma[v\_n := null]$, $\tau' = \tau[v\_n := \tau(v)]$. And, if v_n is already in σ, then we have:

$$\frac{\langle v\_n, * \rangle \in \sigma}{\langle \mathbf{Pre}.dot(v, n), \sigma, \tau \rangle \rightarrow v\_n}, \tag{6}$$

Derivative over other variable u with order n:

$$\langle \mathbf{Pre}.dot(v, u, n), \sigma, \tau \rangle \rightarrow \frac{d^n v}{du^n}, \tag{7}$$

and, if v_y_n is new, then we have

$$\frac{\langle v\_y\_n, * \rangle \notin \sigma}{\langle \mathbf{Pre}.dot(v, y, n), \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.v\_y\_n, \sigma', \tau' \rangle}, \tag{8}$$

otherwise,

$$\frac{\langle v\_y\_n, * \rangle \in \sigma}{\langle \mathbf{Pre}.dot(v, y, n), \sigma, \tau \rangle \rightarrow v\_y\_n}. \tag{9}$$

– **Assignment**.
For single assignment,

$$\langle \mathbf{Pre}.(v = e), \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.skip, \sigma', \tau \rangle, \tag{10}$$

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

$$Discrete()\{x = y; y = x;\}$$

(a) As Sequential Assignment: executing S in a state with $x = 0$ and $y = 1$, x and y are both evaluate to the value 1. For assignment statements S_1, S_2 in Sequential Assignment method,

$$\frac{\langle \mathbf{Pre}.S_1, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.skip, \sigma', \tau \rangle}{\langle \mathbf{Pre}.(S_1; S_2), \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.S_2, \sigma', \tau \rangle}, \tag{11}$$

(b) As Parallel Assignment: executing S in the same state, x and y exchange their value, x is changed to 1, y is 0. For assignment statements S_1, S_2 in Parallel Assignment method, v_1 is the variable modified by S_1 and v_2 of S_2,

$$\frac{\langle \mathbf{Pre}.S_1, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.skip, \sigma', \tau \rangle, \langle \mathbf{Pre}.S_2, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.skip, \sigma'', \tau \rangle}{\langle \mathbf{Pre}.(S_1 || S_2), \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.skip, \sigma''', \tau \rangle,} \tag{12}$$

where, $\sigma''' = \sigma[v_1 := \sigma'(v_1), v_2 := \sigma''(v_2)]$, '||' denotes that the assignments (S_1, S_2) in *Discrete* method of *ParallelAssignment* object are executed in parallel.

– **Method Invocation**.
(a) Zero-Arity-Argument method $m()$:

$$\langle \mathbf{Pre}.m(), \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}'.S, \sigma, \tau \rangle, \tag{13}$$

where, $\mathbf{Pre}' = \mathbf{Pre}.m$ and S is the body of method m.

(b) Fixed-Arity-Argument method $m(arg[1..n])$:

$$\langle \mathbf{Pre}.m(exp[1..n]), \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}'.S, \sigma', \tau' \rangle, \tag{14}$$

where, $\mathbf{Pre}' = \mathbf{Pre}.m(exp[1..n])$ and S is the body of method m, for $1 \leq i \leq n$, $arg[i]$ is a new variable, and,

$$\sigma' = \sigma[arg[i] := \sigma(exp[i])],$$

if $\tau(exp[i])$ is a subtype of the defined type of $arg[i]$, then

$$\tau' = \tau[arg[i] := \tau(exp[i])],$$

otherwise, $\tau'(arg[i])$ takes the defined type of the formal parameter.

– **Instance variable**. Suppose var is an instance variable of the object obj.

(a) Declaration of instance variable without initialization. Consider the declaration D:

$$Type \;\; var;$$

This defines a variable var of type $Type$ and assigns the special value $null$ to var. Thus, we have

$$\langle \mathbf{Pre}.obj.D, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.obj.Skip, \sigma', \tau' \rangle, \tag{15}$$

where, $\sigma' = \sigma[var := null]$ and $\tau' = \tau[var := Type]$.

(b) Declaration of instance variable with initialization. Consider the declaration D:

$$Type \;\; var = val;$$

This defines a variable var of type $Type$ and assigns the value val to var. Thus, we have

$$\langle \mathbf{Pre}.obj.D, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.obj.Skip, \sigma', \tau' \rangle, \tag{16}$$

where, $\sigma' = \sigma[var := val]$ and if $\tau(val)$ is a subtype of $Type$, then $\tau' = \tau[var := \tau(val))]$, otherwise, $\tau' = \tau[var := Type]$.

– **Local variable**. Suppose var is a local variable in the method m or block b. The following are demonstrated under the scenario with method m.

(a) Declaration of local variable without initialization. Consider the declaration D:

$$Type \;\; var;$$

This defines a variable var of type $Type$ and assigns the special value $null$ to var. Thus, we have

$$\langle \mathbf{Pre}.m.D, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.m.Skip, \sigma', \tau' \rangle, \tag{17}$$

where, $\sigma' = \sigma[var := null]$ and $\tau' = \tau[var := Type]$.

(b) Declaration of local variable with initialization. Consider the declaration D:

$$Type \;\; var = val;$$

This defines a variable var of type $Type$ and assigns the value val to var. Thus, we have

$$\langle \mathbf{Pre}.m.D, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.m.Skip, \sigma', \tau' \rangle, \tag{18}$$

where, $\sigma' = \sigma[var := val]$ and if $\tau(val)$ is a subtype of $Type$, then $\tau' = \tau[var := \tau(val))]$, otherwise, $\tau' = \tau[var := Type]$.

(c) End of method m.

$$\langle \mathbf{Pre}.m.End, \sigma, \tau\rangle \rightarrow \langle \mathbf{Pre}.Skip, \sigma', \tau'\rangle, \tag{19}$$

where, $\sigma' = \sigma[return := \sigma(returnExp), rm\ vars]$ and $\tau' = \tau[return := \tau(returnExp), rm\ vars]$. '$rm\ vars$' represents the removing of all the mappings related to local variables of the method m. *End* denotes the end of the method, usually a method is ended by explicitly a *Return* statement or the right brace '}' positioned at the end of the method body. $return$ is the special variable refers to the result of the method invocation, $returnExp$ denotes the value of the variable. And, for block b, the special variable $return$ is ignored.

– **Object Creation**. The procedure of object creation is composed of instance variable initialization and constructor invocation.

(a) Creation by Constructor. If the object creation statement S is

$$Type\ obj = new\ M(exp[0..n]);$$

where, $exp[0..n]$ represents the list (or array) of actual parameters. M is the name of the instantiated class, also the name of the constructor, $M(exp[0..n])$ is an invocation of the corresponding constructor in the class. Suppose the set of instance variable declaration is Ds, and constructor $m(arg[0..n])$,

$$\langle \mathbf{Pre}.S, \sigma, \tau\rangle \rightarrow \langle \mathbf{Pre}'.(Ds; M(exp[0..n])), \sigma', \tau'\rangle, \tag{20}$$

where, $\mathbf{Pre}' = \mathbf{Pre}.obj$, $\sigma' = \sigma[obj := o]$, o is a new object of $Type$ with all the instance variables refer to the special value $null$, $\tau' = \tau[obj := Type]$.

(b) Creation by Anonymous Class. If the object creation statement S is

$$Type\ obj = new\ Identifier(){Class\ Body};$$

Then,

$$\langle \mathbf{Pre}.S, \sigma, \tau\rangle \rightarrow \langle \mathbf{Pre}'.(Ds; M()), \sigma', \tau'\rangle, \tag{21}$$

where, it is the same as (20) except that it executes the zero-arity-argument constructor. If there is no zero-arity-argument constructor declares in the class body, the empty one would take the job that doing nothing when it is invoked. The empty constructor is implicitly declared in one class for the case that the zero-arity-argument constructor is missing by the designer.

– **Dynamic**. In Apricot, the *Dynamic* object consists of one *Continuous* method and an *Invariant* block. The *Continuous* method declares the differential equations that the dynamic flow followed with respect to the properties defined within the *Invariant* block. The properties in the *Invariant* block indicate the range of the variables during the continuous evolution. For dynamic, if the dynamic flow reaches the border of the *Invariant* and all the conditions of the compositions from the dynamic can not be satisfied, then the control is waiting at the border provided that any advancement of the flow according to the *Continuous* method will violate the *Invariant*.

(a) Differential Equation. For one statement D that is declared in the $Continuous$ method, D is a differential equation for the variable v. For variable v and nature number n, mathematic expression me, the differential equation D is

$$dot(v, n) == me;$$

Suppose that there exists a function $f : I \rightarrow \mathbb{R}$, and I is a time-interval $[a, b]$, i.e., the domain of f, and the value of v at time-point $t \in [a, b]$ is $f(t)$. Here, the

start time-point of the continuous evolution following D is at time a, the end point b is for some proper time-point greater than or equals a. Then, before the termination of the flow, at some time-point $t \in [a, b]$, we have

$$\langle \mathbf{Pre}.D, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.D, \sigma', \tau \rangle, \tag{22}$$

where, $\sigma' = \sigma[v := f(t)]$. We call f the *Real-Function* for D, and t the *Proper-Time*.

(b) Termination of Flow. The dynamic flow reaches the border of the *Invariant* and no valid composition relationship exists, then the control is waiting at the border if the forward flow would violate the *Invariant*.

$$\frac{\begin{array}{c}\forall c \in C, \langle \mathbf{Pre}.c, \sigma, \tau \rangle \rightarrow False, \\ \langle \mathbf{Pre}.D, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.D, \sigma', \tau \rangle, \exists i \in I, \langle \mathbf{Pre}.i, \sigma', \tau \rangle \rightarrow False\end{array}}{\langle \mathbf{Pre}.D, \sigma, \tau \rangle \rightarrow \langle \mathbf{Pre}.(dot(tw, 1) = 1), \sigma, \tau \rangle}, \tag{23}$$

where, the set C is the *Condition* block related to the current *Dynamic* object that contains the differential equation D. And, I is the *Invariant* block in the *Dynamic* object, it is the set of conditions should be satisfied during the continuous evolution. For $\forall t \in (a, b]$, $\sigma' = \sigma[v := f(t)]$, in which, $f : I \rightarrow \mathbb{R}$, $I = [a, b]$, $f(t)$ is the value of v at the time-point $t \in I$. At last, tw is a specific variable for the waiting time after the flow terminated.

– **Invariant**. An Invariant block I is a built-in block in a *Dynamic* object. Actually, I consists of conditions. Each condition specifies the range of one variable, and can be evaluated as a Boolean expression. Suppose $i \in I$, and $i \equiv v \ in \ (e_1, e_2)$, we have

$$\frac{\langle \mathbf{Pre}.(e_1, e_2), \sigma, \tau \rangle \rightarrow (n_1, n_2), \sigma(v) \in (n_1, n_2)}{\langle \mathbf{Pre}.(v \ in \ (e_1, e_2)), \sigma, \tau \rangle \rightarrow True}, \tag{24}$$

where, v takes the value in the interval denoted by (e_1, e_2). And, the opposite situation,

$$\frac{\langle \mathbf{Pre}.(e_1, e_2), \sigma, \tau \rangle \rightarrow (n_1, n_2), \sigma(v) \notin (n_1, n_2)}{\langle \mathbf{Pre}.(v \ in \ (e_1, e_2)), \sigma, \tau \rangle \rightarrow False}. \tag{25}$$

Now, we have the evaluation of an Invariant I based on the up two laws,

$$\frac{\forall i \in I, \langle \mathbf{Pre}.i, \sigma, \tau \rangle \rightarrow True}{\langle \mathbf{Pre}.I, \sigma, \tau \rangle \rightarrow True}, \tag{26}$$

where, I is true when all the conditions in it is true. And, if there exists an invalid condition, then I is false,

$$\frac{\exists i \in I, \langle \mathbf{Pre}.i, \sigma, \tau \rangle \rightarrow False}{\langle \mathbf{Pre}.I, \sigma, \tau \rangle \rightarrow False}. \tag{27}$$

– **Condition**. A *Condition* block C consists of a number of Boolean expressions. Each Boolean expression c involves two mathematic expressions (me_1, me_2) and a relational operator *opt*. Let $c \equiv me_1 \ opt \ me_2$, $opt \in \{==, <, >, <=, >=, !=\}$, and C for the set of all Boolean expressions in the *Condition* block,

$$\frac{\forall c \in C, \langle \mathbf{Pre}.c, \sigma, \tau \rangle \rightarrow True}{\langle \mathbf{Pre}.C, \sigma, \tau \rangle \rightarrow True}, \tag{28}$$

where, C is true iff all Boolean expressions in C is true.

– **Composition Relationship**. It involves the control switch from one dynamic to another under proper conditions. Let D_1, D_2 represent two *Dynamic* objects, they may be the same object, e.g., in Example 1. And, 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,

$$\frac{\langle \mathbf{Pre}.C, \sigma, \tau\rangle \to True, \langle \mathbf{Pre}.A, \sigma, \tau\rangle \to \langle \mathbf{Pre}.Skip, \sigma', \tau\rangle, \langle \mathbf{Pre}.D_2.I, \sigma', \tau\rangle \to True}{\langle \mathbf{Pre}.R(D_1, A, D_2, C), \sigma, \tau\rangle \to True}, \quad (29)$$

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

$$\frac{\langle \mathbf{Pre}.R(D_1, A, D_2, C), \sigma, \tau\rangle \to True}{\langle \mathbf{Pre}.D_1, \sigma, \tau\rangle \to \langle \mathbf{Pre}.D_2, \sigma', \tau\rangle}. \quad (30)$$

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 to violate the *Invariant*, then the choice to switch or continue the flow itself is nondeterministic.

– **Start Dynamics**. For *Dynamics* D_1 and D_2, the composite for start statements, is the parallel evolution of the continuous flows, let

$$D_1||D_2 \equiv D_1.start(); D_2.start(),$$

then, we have

$$\frac{\langle \mathbf{Pre}.D_1, \sigma, \tau\rangle \to \langle \mathbf{Pre}.D_1, \sigma_1, \tau\rangle, \langle \mathbf{Pre}.D_2, \sigma, \tau\rangle \to \langle \mathbf{Pre}.D_2, \sigma_2, \tau\rangle}{\langle \mathbf{Pre}.(D_1||D_2), \sigma, \tau\rangle \to \langle \mathbf{Pre}.(D_1||D_2), \sigma', \tau\rangle}, \quad (31)$$

where, $\sigma_1 = \sigma[v_1 := f_1(t)]$, and $\sigma_2 = \sigma[v_2 := f_2(t)]$, therefore,

$$\sigma' = \sigma_1[v_2 := f_2(t)] = \sigma_2[v_1 := f_1(t)] = \sigma[v_1 := f_1(t), v_2 := f_2(t)].$$

Here, f_1, f_2 are the *Real-Function*s for D_1 and D_2, respectively. And, t is the *Proper-Time*.

– **Parallel Composition Relationship**. For two composition relationships CR_s and CR_t, the parallel composition relationship is defined as follows,

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

First, we have the parallel execution of *Assignment* objects A_s and A_t,

$$\frac{\langle \mathbf{Pre}.D_{s_1}, \sigma, \tau\rangle \to \langle \mathbf{Pre}.D_{s_2}, \sigma', \tau\rangle, \langle \mathbf{Pre}.D_{t_1}, \sigma', \tau\rangle \to \langle \mathbf{Pre}.D_{t_2}, \sigma'', \tau\rangle}{\langle \mathbf{Pre}.(A_s||A_t), \sigma, \tau\rangle \to \langle \mathbf{Pre}.Skip, \sigma'', \tau\rangle}, \quad (32)$$

where, A_s and A_t are symmetrical. The parallel composition relationship is valid,

$$\frac{\begin{array}{c}\langle \mathbf{Pre}.(CR_s \; and \; CR_t), \sigma, \tau\rangle \to True, \\ \langle \mathbf{Pre}.(A_s||A_t), \sigma, \tau\rangle \to \langle \mathbf{Pre}.Skip, \sigma'', \tau\rangle, \langle \mathbf{Pre}.(D_{s_2}.I \; and \; D_{t_2}.I), \sigma'', \tau\rangle \to True,\end{array}}{\langle \mathbf{Pre}.(CR_s||CR_t), \sigma, \tau\rangle \to True}, \quad (33)$$

where, the Boolean operator *and* represents the conjunction relation.

4 Design by Convention

Design by convention is a software design paradigm that is known as convention over configuration (abbreviated as COC). It evicts the decisions the developers need to make by the conventional usages of the design ingredients, given the simplicity during the modeling process. In software development, COC is usually used for the least configuration that the developer should to set down. We apply the idea of COC and utilize it in the design of hybrid systems, and name it as design by convention (abbreviated as DBC) in our language.

4.1 The composition of statements

For boolean expressions A and B,

$$Condition\{A; B; \}; \equiv A \wedge B.$$

We do not need to explicitly add the conjunction operation to connect the boolean expressions, the separate expressions in the *Condition* block have the conjunction relationship implicitly. It also makes the conditions more clear and be easy to understand.

For the parallel and sequential assignments, they have the same appearance, but, different execution semantics indicated by the different Interfaces.

$$ParallelAssignment\{Discrete()\{A; B; \}\}; \equiv A||B,$$
$$SequentialAssignment\{Discrete()\{A; B; \}\}; \equiv A; B.$$

The implementation of Interface *ParallelAssignment* gives the statements A and B the parallel composition relationship. While, the sequential composition of A and B is prominent for the case of Interface *SequentialAssignment*.

In a similar way, the starts of dynamics in the *Initializer* method for the *System* class have the parallel composition semantics without to employ the parallel operator '||'.

$$Init\{A.start(); B.start(); \}; \equiv A||B$$

And, in the constructor of a *System* class, we can ignore the parallel indications for plants and controllers if they have the starts of dynamics in the *Initializer*. For instance, the '`god||ball`' in Fig.1(f) can be wiped off.

4.2 The inexistence

For True Condition and Invariant,

$$Condition\{\}; \equiv True, Invariant\{\}; \equiv True.$$

We evaluate the empty *Condition* and *Invariant* blocks to *True*, and the inexistent of these two blocks also considered to the boolean *True*.

For *Empty* assignment or the non-initialization of the assignment instance variable, we evaluate it to the special statement *Skip*.

$$Comp(Dy_1, , Dy_2) \equiv Comp(Dy_1, Skip, Dy_2),$$

where Dy_1 and Dy_2 are dynamics and the ' ' (Blank Space) in the LHS denotes the empty assignment.

5 Conclusions

In this paper, we proposed Apricot as an object-oriented language for modeling hybrid systems and 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, as a result, bring about a modeling language with simple and distinct syntax, structure and semantics. We also discussed the design by convention features of Apricot. For the future work, we will focus on the formal verification for Apricot models, then investigate verification techniques and develop relevant tools.

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., Henzinger, T., Ho, P.: Automatic symbolic verification of embedded systems. IEEE Transactions on Software Engineering 22(3), 181–201 (1996)
3. America, P., de Bakker, J., Kok, J., Rutten, J.: Operational semantics of a parallel object-oriented language. In: Proceedings of POPL'86. pp. 194–208. ACM (1986)
4. Apt, K., De Boer, F., Olderog, E., de Gouw, S.: Verification of object-oriented programs: a transformational approach. Journal of Computer and System Sciences (2011)
5. 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)
6. Cuijpers, P.J.L., Reniers, M.A.: Hybrid process algebra. The Journal of Logic and Algebraic Programming 62(2), 191–245 (2005)
7. Dorf, R.C., Bishop, R.H.: Modern Control Systems. Prentice Hall (2011)
8. Fowler, M.: Domain-specific languages. Addison-Wesley Professional (2010)
9. Frehse, G.: Phaver: algorithmic verification of hybrid systems past hytech. International Journal on Software Tools for Technology Transfer 10(3), 263–279 (2008)
10. 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)
11. 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)
12. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design patterns: Abstraction and reuse of object-oriented design. In: Proceedings of ECOOP'93. LNCS, vol. 707, pp. 406–431. Springer-Verlag (1993)
13. He, J.: From csp to hybrid systems, a classical mind: essays in honour of car hoare (1994)
14. He, J., Li, X., Liu, Z.: rcos: A refinement calculus of object systems. Theoretical Computer Science 365(1), 109–142 (2006)
15. 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)
16. Platzer, A.: Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics. Springer (2010)
17. Platzer, A., Quesel, J.D.: Keymaera: A hybrid theorem prover for hybrid systems (system description). In: Automated Reasoning, pp. 171–178. Springer-Verlag (2008)
18. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Department of Computer Science, Aarhus university (September 1981)
19. Plotkin, G.D.: A structural approach to operational semantics. The Journal of Logic and Algebraic Programming 60-61, 17–139 (2004)
20. 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)
21. 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)

A Identifiers

An *identifier* is an unlimited-length (but the length is greater than one) sequence of letters and digits, but not a Keyword:

$$\begin{aligned} Letter &::= \texttt{a} \mid \texttt{b} \mid ... \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid ... \mid \texttt{Z}; \\ Digit &::= \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \texttt{0}; \\ ValidChar &::= Letter \mid Digit; \\ Identifier &::= Letter\{Letter \mid Digit\}^*. \end{aligned}$$

In which the letter is defied as the character in the set {`a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`, `j`, `k`, `l`, `m`, `n`, `o`, `p`, `q`, `r`, `s`, `t`, `u`, `v`, `w`, `x`, `y`, `z`, `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J`, `K`, `L`, `M`, `N`, `O`, `P`, `Q`, `R`, `S`, `T`, `U`, `V`, `W`, `X`, `Y`, `Z` }.

B Types, Values, and Variables

The types of the Apricot language are divided into two categories: mathematic types and reference types.

$$\begin{aligned} Type ::= & PrimitiveType \mid MathematicType \\ & \mid ReferenceType; \end{aligned}$$

B.1 Mathematic Types and Values

Primitive Type is the same as mathematicType except that, primitive type variable can not be shared and has the feature of "call-by-value" during method calls. Call-by-value requires the evaluation of the arguments before passing them to the definition of the method. Another style is call-by-name which passing the arguments directly to the definition. For mathematic and reference types we take the call-by-name style argument passing for method invocation. In addition, there is a difference between mathematic type and reference type. Reference type variables can refer to another object with the same type by the assignment statement. But, the assignment can only change the mathematical value of the object for mathematic type variables. It means that, when a mathematic type variable refers to a methematic type object for the first time, the variable will hold this object all the time and only the mathematical value of this object can be updated.

$$\begin{aligned} MathematicType &::= NumbericType \mid \texttt{Boolean}; \\ NumbericType &::= \texttt{Integer} \mid \texttt{Real}; \end{aligned}$$

Accordingly, the primitive type is defined by:

$$PrimitiveType ::= \texttt{integer} \mid \texttt{real} \mid \texttt{boolean};$$

1. Mathematic types : `Boolean` type and the numeric types. The `Boolean` type represents a logical quantity in the literals set { `True`, `False`}. The numeric types are the integer type `Integer`, and the real number type `Real`;
2. Reference types : class types, interface types, and array types.

An object is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to objects.

B.2 Reference Types and Values

There are four kinds of reference types: class types, interface types, type variabless, and array types.

$$\begin{aligned} ReferenceType ::= & ClassType \mid InterfaceType \\ & \mid ArrayType; \\ ClassType ::= & Identifier; \\ InterfaceType ::= & Identifier \mid \texttt{System} \mid \texttt{Plant} \\ & \mid \texttt{Controller} \\ & \mid \texttt{Dynamic} \mid \texttt{Assignment} \\ & \mid \texttt{ParallelAssignment} \\ & \mid \texttt{SequentialAssignment} \\ ArrayType ::= & Type\ [\ ]. \end{aligned}$$

B.3 Variables

A variable is a physical quantity name in physical world or a storage location in the memory of computer, and has an associated type that is either a mathematic type or a reference type.

The value of a variable is changed by an assignment or according to the differential equations defined in `Dynamic` classes.

For all types, the default value of any type variable is the special value `null`.

B.4 Variables of Mathematic Type

Mathematic type variables are always hold a mathematic value of that exact mathematic type.

B.5 Variables of Reference Type

A variable of a reference type `R` can hold a null reference, a reference to an instance of class `C`, any class that is a subclass of `C`, any class that is a implementation of interface `C` or any array type.

C Mathematical Operations

C.1 Arithmetic Operators

For $x, y \in \mathcal{R}$, the following arithmetic operators are defined on Real numbers ($\mathcal{R}$):

1. $x + y$, binary plus, addition;
2. $x - y$, binary minus,subtraction;
3. $x * y$, binary multiple, multiplication;
4. x/y, binary divide, division;
5. $+x$, unary plus, it denotes the identity operation on x, thus, $x == +x$ with respect to the evaluation;
6. $-x$, unary minus, inverse operation on x, thus, $(-x) + x == 0$.

C.2 Boolean Operators

Standard boolean operators are defined for all `Boolean` type values x, y:

1. ==, equality;
2. ! =, inequality;
3. !, logical complement;
4. `in`, belong to interval, the result value of (x `in` (a, b)) is `True` iff $a < x < b$, (x `in` $[a, b]$) is `True` iff $a \leq x \leq b$, (x `in` $(a, b]$) is `True` iff $a < x \leq b$, and (x `in` $[a, b)$) is `True` iff $a \leq x < b$;
5. `and`, the result value of (x `and` y) is `True` if both operand values are `True`;
6. `xor`, the result value of (x `xor` y) is `True` if the operand values are different;
7. `or`, the result value of (x `or` y) is `True` if one of the operand values is `True`.

C.3 Numeric Comparisons

Standard comparison operations are defined for all Real numbers ($\mathcal{R}$), which result in a value of type `Boolean`:

1. ==, equality;
2. ! =, inequality;
3. <, less than;
4. <=, less than or equal to;
5. >, greater than;
6. >=, greater than or equal to.

Special Symbol numbers:

1. Inf is stands for ∞, which is equal to itself and greater than any other number;
2. $-Inf$ is stands for $-\infty$, which is equal to itself and less then any other number;

C.4 Mathematical Functions

We provides a comprehensive collection of mathematical functions and operators. These mathematical operations are defined on Real numbers ($\mathcal{R}$).

1. $dot(x, n)$, n-th order derivative of x over time (t), i.e. $dot(x, n) = \frac{d^n x}{dt^n}$.
2. $dot(x, y, n)$, n-th order derivative of x over y, i.e. $dot(x, y, n) = \frac{d^n x}{dy^n}$.
3. Standard trigonometric functions: sin, cos, tan, cot, sec and csc.
4. $round(x)$, round x to the nearest integer, omitting decimal fractions smaller than 0.5, e.g. $round(2.5) = 3$, $round(0.4) = 0$.
5. $floor(x)$, round x towards $-Inf$, e.g. $round(2.5) = 2$.
6. $ceil(x)$, round x towards $+Inf$, e.g. $ceil(2.5) = 3$.
7. $div(x, y)$, truncated division, and quotient rounded towards zero.
8. $fld(x, y)$, floored division, quotient rounded towards $-Inf$.

9. $rem(x, y)$, remainder, satisfies $x = div(x, y) * y + rem(x, y)$, implying that sign of $rem(x, y)$ matches x.
10. $mod(x, y)$, modulus; satisfies $x = fld(x, y) * y + mod(x, y)$, implying that sign of $mod(x, y)$ matches y.
11. $gcd(x_1, x_2, ..., x_n)$, greatest common divisor of x_1, x_2, ..., x_n with sign matching x_1.
12. $lcm(x_1, x_2, ..., x_n)$, least common multiple of x_1, x_2, ..., x_n with sign matching x_1.
13. $abs(x)$, a positive value with the magnitude of x.
14. $sign(x)$, indicates the sign of x, returning -1, 0, or $+1$.
15. $sqrt(x)$, the square root of x, i.e. x^2.
16. $root(x, b)$, the b-th root of x, i.e. $\sqrt[b]{x}$.
17. $hypot(x, y)$, accurate $sqrt(x^2 + y^2)$ for all values of x and y.
18. $pow(x, y)$, x raised to the exponent y, i.e. x^y.
19. $exp(x)$, the natural exponential function at x, i.e. e^x.
20. $log(x)$, the natural logarithm of x, i.e. $\log(x)$ or $\ln(x)$.
21. $log(b, x)$, the base b logarithm of x, i.e. $\log_b(x)$.
22. $erf(x)$, the error function (Gauss error function) at x, i.e. $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} dt$.
23. $gamma(x)$, the gamma function at x.
24. $max(x_1, ..., x_n)$.
25. $min(x_1, ..., x_n)$.