The TIOA Language
Version 0.21

Dilsun Kaynar, Nancy Lynch, Sayan Mitra, Stephen Garland

May 22, 2005

1 Introduction

The timed input/output automaton (TIOA) framework [3, 4] is a mathematical modeling framework that supports the description and analysis of timed systems. The TIOA language is a formal language for specifying timed systems and their properties within the TIOA framework.

The TIOA language is a variant of the IOA language [2], which supports the description of basic I/O automata with no timing information. The TIOA language does not provide the full generality of the mathematical TIOA framework [3] with respect to defining dynamic types and trajectories; for example, it restricts differential equations in trajectory definitions to those in which $d(v)$ equals the value of some expression.

2 TIOA Language

The TIOA language is based on the IOA language, but extends, simplifies, and differs from it in the following respects.

- TIOA contains notations for time-related aspects of timed I/O automata: both discrete and continuous types, stopping conditions for transitions, and trajectories.
- In TIOA, notations for operators on datatypes are defined using a new **vocabulary** construct. In IOA, such notations were defined in auxiliary files, written in the Larch Shared Language (LSL). This change has two advantages:
  - Complete TIOA specifications can be written in a single file and can be checked statically without the need for auxiliary files. (TIOA specifications can still be distributed over several files, if the so user wishes).
  - TIOA is more amenable than IOA for use with theorem provers (e.g., PVS) other than the Larch Prover. It allows axioms for abstract data types to be expressed in languages appropriate for those theorem provers, rather than requiring them to be written in LSL.
  - Definitions of automata and vocabularies use the same parameterization mechanism (in IOA, auxiliary LSL specifications used a different mechanism).
- TIOA contains notations for defining functions and derived variables.

- TIOA contains notations for new NDR (nondeterminism resolution) constructs. These are used to provide scheduling information for the TIOA simulator.
- TIOA contains notations for guiding the (paired) simulation of trajectories.

This paper defines the syntax and semantics of TIOA using the conventions of the IOA Manual [2]. The syntax of TIOA is the same as that of IOA, unless explicitly stated otherwise. The syntax and semantics of language constructs not explained in this paper can be found in [2].

2.1 Vocabularies

TIOA does not use any of the IOA keywords specific to LSL, nor does it use the keyword **axioms**. Instead of using an **axioms** statement to refer to separate LSL specifications containing definitions of types and operators, it uses a **vocabulary** definition to introduce **types** and **operators**.

2.1.1 Surface Syntax

```
spec            ::=  (funcDecls | vocabDef | automatonDef | assertion)+
vocabDef        ::=  'vocabulary' vocabId formals?  defines?
                     imports?  typeDcls?  opDcls?
vocabId              IDENTIFIER
formals         ::=  '(' formal,+ ')'
formal          ::=  IDENTIFIER,+ ':'  (type | 'type')

defines         ::=  'defines' constructorDcl
imports         ::=  'imports' vocabRef,+
vocabRef        ::=  vocabId actuals?
actuals         ::=  '(' actual,+ ')'
actual          ::=  term | 'type' type

typeDcls        ::=  'types' typeDcl,*
typeDcl         ::=  type shorthand?
constructorDcl  ::=  typeConstructor '[' simpleType+ ']'

opDcls          ::=  'operators' opDcl,*
opDcl           ::=  name,+ ':'  signature ','?
```

The *vocabulary* of a `vocabDef` contains

- each `type`, `typeConstructor`, and `operator` defined in its `defines`, `typeDcls`, and `opDcls`,

- the vocabulary of each `shorthand` in its `typeDcls`, and
- the appropriately renamed vocabulary of each imported `vocabRef`.[1]

The *base vocabulary* of a `spec` is the union of the vocabularies of TIOA's built-in types. (This vocabulary is simply called the vocabulary of the specification in [2].

2.1.2 Intermediate Language Syntax

```
vocabulary  ::=  '(' 'vocab' vocabName formals?  defines?
                 imports?  sortDcls?  opDcls?  ')'
vocabName   ::=  identifier
formals     ::=  '(' 'formals' formal+ ')'
defines     ::=  '(' 'defines' sortId ')'
imports     ::=  '(' 'imports' vocabRef+ ')'
sortDcls    ::=  '(' 'sorts' sort+ ')'
opDcls      ::=  '(' 'ops' operator+ ')'
```

2.1.3 Static Semantics

The `simpleTypes` in a `constructorDcl` must be distinct.

A vocabulary cannot have both `formals` and a `defines` clause.

Each `type` in a `formal` or `signature` must be in the vocabulary of the `vocabDef`.

More to come.

2.2 Function Definitions

A new construct enables the definition of functions at various points in a TIOA specification. Such definitions can occur as a specification unit (analogous to `vocabDef`s and `automatonDef`s), following the definition of state variables (see Section 2.3), or at the beginning of a transition or trajectory definition (see Sections 2.5 and 2.6).

2.2.1 Surface Syntax

```
funcDecls  ::=  'let' funcDecl;+
funcDecl   ::=  funcName '(' variableList,+ ')' '=' term
funcName   ::=  IDENTIFIER
```

The domain of the function defined in a `funcDecl` consists of the sequence of types associated with the variables in its `variableLists`, and the range of the function is the type of the defining `term`.

[1]To regain functionality present in IOA and LSL, the definitions of `formal` and `actual` may be enhanced to allow type constructors and operators, and the definition of `actuals` may be enhanced to allow renamings of types, type constructors, and operators that do not appear as **formals**.

2.2.2 Intermediate Language Syntax

```
funcDecls  ::=  '('  'let' funcDecl+ ')'

funcDecl   ::=  '('  opId varId* term ')'
```

2.2.3 Static Semantics

Each variable in a `variableList` of a `funcDecl` must be distinct from all other such variables, that is, they must differ either in their `IDENTIFIER`s or their `type`s.

The type of each such variable must be in the base vocabulary of the specification or, if the `funcDecl` occurs within an automaton definition, in the vocabulary of that automaton.

When a `funcDecl` occurs outside an automaton definition, each constant, operator, or variable in its defining `term` must be in the base vocabulary of the specification or declared in a `variableList` in the `funcDecl`.

When a `funcDecl` occurs within an automaton definition, each variable in its `variableList`s must differ from any `automatonFormal` and from any `stateVariable`; furthermore, each constant, operator, or variable in its defining `term` must be in the vocabulary of the automaton or declared in a `variableList` in the `funcDecl`.

When a `funcDecl` occurs within a transition or trajectory definition, each variable in its `variableList`s must differ from any parameter of that transition or trajectory definition; furthermore, each constant, operator, or variable its defining `term` must be in the vocabulary of the automaton, declared in a `variableList` in the `funcDecl`, or a parameter of that transition or trajectory.

2.3 Primitive Automaton Definitions

A primitive TIOA is defined by its action signature, its states, its transitions, its trajectories, and its tasks. Trajectory, task, and schedule specifications are optional.

The naive translation of any IOA program is a legal TIOA program provided that the variables of type **Real** are interpreted as of new built-in type **DiscreteReal**.

2.3.1 Surface Syntax

```
automatonDef    ::=  'automaton' automatonName formals?
                     imports?  (basicAutomaton | composition)

basicAutomaton  ::=  'signature' formalActions+ states funcDecls?
                     transitions trajectories?  tasks?  schedule?
```

2.3.2 Intermediate Language Syntax

```
automaton  ::=  '('  'automaton' automatonName  formals where?  ?
                     imports?  automatonDef ')'
primitive  ::=  '('  '(' 'actions' action+ ')' states funcDecls?
                     ( '(' 'transitions transition+ ')' )?
                     ( '(' 'trajectories trajectory+ ')' )?
                     ( '(' 'tasks' task+ ')' )?
                     schedule?  ')'
```

2.3.3 Semantics

Section 2.6 defines the semantics for trajectory definitions.

2.4 State Variables

State variables in TIOA are declared exactly as in IOA. Static types in TIOA correspond to types in IOA. Dynamic types in TIOA have no counterpart in IOA. Dynamic types of variables in TIOA are declared implicitly (and can be inferred by TIOA implementations automatically) for all built-in simple types and for a predefined class of compound types as explained below. The dynamic type of all other built-in and user-defined types consists of the piecewise constant functions.

State variables may have a new built-in type **AugmentedReal** for the set $R \cup \{\infty, -\infty\}$.[2]

2.4.1 Semantics

- If v is a variable of type **Real** or **AugmentedReal**, then the dynamic type of v ($dtype(v)$) is the pasting closure of the set of continuous functions from left-closed intervals of time to the set of reals (piecewise continuous functions). If v is a variable of type **DiscreteReal** or **DynamicAugmentedReal**, then $dtype(v)$ is the pasting closure of the set of constant functions from left-closed intervals of time to the set of reals (piecewise constant functions).

- If v is a variable of any other simple type, then $dtype(v)$ is the pasting closure of the set of constant functions from left-closed intervals of time to the set of reals.

- If v is a variable of one of the following compound types:

  1. **Array[I1, . . . , In,E]**: n-dimensional array of elements of type **E** indexed by element of types **I1, . . . , In** where the indices are of finite types.

[2]The **AugmentedReal** trait permits comparisons of real numbers with augmented real numbers; it also permits arithmetic operators to be applied to one real and one augmented real number, so there is no need to cast reals explicitly to augmented reals. Similarly, it is also possible to apply arithmetic operators to arguments that have either discrete or continuous types, with the result type being discrete if both arguments are discrete.

Eventually, TIOA may allow users to define dynamic types, e.g., by defining a topology (or metric) on the set of elements in a data type and using that topology (or metric) for the definition of continuity. With this approach, piecewise constant functions are the same thing as piecewise continuous functions in a discrete topology. Hence, it might make more sense to have the default dynamic type be piecewise continuous and the default toplogy be the discrete topology.

2. **Map[D1, ..., Dn,E]**: finite partial mapping of an n-dimensional domain with type **D1** $\times$ **...** $\times$ **Dn** to elements of a range with type **E**. Mappings differ from arrays in that they are defined only for finitely many elements of their domains (and hence may not be totally defined).
3. **Tuple of I1: E1, ... In: En**: n-tuple with fields **I1, ..., In** with types, respectively, **E1, ... En**.
4. **Seq[E]**: finite sequence of elements of type E.

then $dtype(v)$ is defined recursively as follows. All of the above-listed datatypes can be viewed as functions from a domain Dom such that $Dom = \{v_1, \ldots, v_k\}$ for some finite integer k. Then, $dtype(v)$ is defined as the set of functions f from left-closed intervals of time to $type(v)$ such that $f.v_i \in dtype(v_i)$ for each $i \in \{1, \ldots k\}$. We use $f.v_i$ to denote the function that gives $f(t)(v_i)$ for all t in the domain of f.

Example 2.1 Consider the following type definition in TIOA.

```
type T tuple [a: Real, b: Nat, c: DiscreteReal]
```

Suppose that variable v is declared to be of type **T**. Then $dtype(v)$ is the set of functions f from left-closed intervals of time to T such that $f.a$ is a piecewise continuous function with real values, $f.b$ is a piecewise constant functions with natural values, and $f.c$ is a piecewise constant function with real values. ■

Example 2.2 Consider the following type definitions in TIOA. The type **Matrix** represents a two-dimensional array of integers. [[**Steve: It is not currently possible to define Matrix in this way.**]]

```
vocabulary matrices
  types Row enumeration [p1, p2, p3],
        Column enumeration [q1, q2, q3],
        Matrix Array[Row,Column,Int]
```

Suppose that variable v is declared to be of type **Matrix**. The objects of type **Matrix** can be viewed as functions from the set **{p1, p2, p3}** to a set of elements of type **Array[Column,Int]**. Each of these elements in turn can be viewed as a function from the set **{q1, q2, q3}** to the set of integers. The dynamic type of **Matrix** is interpreted recursively as explained above. ■

Example 2.3 The automaton `A(u1, u2: Real)` below, the code of which has not been provided in full, has three state variables: `continue`, `now`, and `deadlinea`. By the semantic rules, the dynamic type of `continue` is the pasting closure of the set of constant functions from left closed intervals of time to the set of booleans, and the dynamic type of `now` is the pasting closure of the set of continous functions from left closed intervals of time to the reals. The type of `deadlinea` gives us specific information about the dynamic type of `deadlinea`. We know not only that they are piecewise continuous but also that they are piecewise constant.

```
 automaton A(u1, u2:Real)
   signature
     internal a, end
   states
```

```
    continue: bool := true,
    now: Real := 0,
    deadlinea: DiscreteReal := u1

  transitions
    internal a
          % omitted from this example
    internal end
          % omitted from this example
  trajectories
          % omitted from this example
```

■

2.5 Transition Definitions

Transition definitions in TIOA can contain urgency predicates.

2.5.1 Surface Syntax

Urgency predicates in TIOA transition definitions begin with the new keywords **urgent when**.

```
transitions    ::=  'transitions' transition+
transition     ::=  actionHead funcDecls? precondition? urgency? effect?
actionHead     ::=  actionType actionName (actionActuals where?)?
actionActuals  ::=  '(' term,+ ')'
precondition   ::=  'pre' predicate
urgency        ::=  'urgent' 'when' predicate
```

Transition definitions for input actions cannot have urgency predicates associated with them.

2.5.2 Intermediate Language Syntax

```
transition  ::=  '('  transitionId caseName actionId
                      ( actionActuals locals where?  )?
                      funcDecls?  precondition?  urgency?  effect?  ')'
urgency     ::=  '('  'urgent' predicate ')'
```

2.5.3 Semantics

Consider a transition definition with a parameterized action name a. Let h be a set of parameter values for a satisfying the where clause, and let b be the action corresponding to the pair (a, h). We call the urgency predicate associated with b, $urgent(b)$ and the precondition associated with b, $pre(b)$. If there is no urgency predicate specified for b, then we assume that $urgent(b) = false$ and if there is no precondition specified for b, then we assume that $pre(b) = true$. The complete definition of the semantics of urgency predicates is given in 2.6.

[[**Dilsun: The code is explained in full when its specification is given in full. That is, in section on trajectories. Move it to here???**]]

Example 2.4 The automaton `A(u1, u2: Real)` below has two transition definitions. The action `a` is enabled when the flag `continue` is set to true and `now` is not equal `u2`. The urgency predicate of `a` implies that if `a` is enabled and `now` is equal to `deadlinea`, then time may not advance before this or some other action has occured. The variable `deadlinea` is incremented by `u1` as a result of action `a`. The transition definition for action `end` has no urgency predicate specified. It is enabled whenever `continue` is set to true and `now` becomes equal to `u2`. The action `end` sets `continue` to false when it is performed.

```
automaton A(u1, u2: Real)
  signature
    internal a, end
  states
    continue: bool := true,
    now: Real := 0,
    deadlinea: DiscreteReal := u1
  transitions
    internal a
        pre continue ∧ now ≠ u2
        urgent when now = deadlinea
        eff deadlinea := deadlinea + u1
    internal end
        pre continue ∧ now = u2
        eff continue := false
  trajectories % omitted from this example
```

■

2.6 Trajectory Definitions

Trajectory definitions specify the trajectories of a TIOA by using differential and algebraic equations, invariants, and stopping conditions. Since IOA has no language construct for specifying trajectories, most of the constructs introduced in this section are new additions to the IOA language. They have been derived from the language constructs that appear in [5, 6, 3].

2.6.1 Surface Syntax

Trajectories are specified by a list of trajectory definitions each of which consists of a name, an optional invariant, an optional stopping condition, and a set of algebraic and differential equations and inequalities, which may written with the aid of function definitions.

```
trajectories  ::=  'trajectories' trajectory+
trajectory    ::=  trajHead funcDecls?  invariant?  stopCond?  evolve
trajHead      ::=  'trajdef' trajDefName
trajDefName   ::=  IDENTIFIER
invariant     ::=  'invariant' predicate
stopCond      ::=  'stop' 'when' predicate
evolve        ::=  'evolve' evolvePred;+
evolvePred    ::=  evolveValue ('=' | comparison) term
              ::=  | term comparison (evolveValue | term)
evolveValue   ::=  lvalue | 'd' '(' lvalue ')'
comparison    ::=  '<' | '<=' | '>' | '>='
```

2.6.2 Intermediate Language Syntax

```
trajectory  ::=  '('  trajId IDENTIFIER funcDecls?
                      ( '(' 'invariant' predicate ')' )?
                      ( '(' 'stop' predicate ')' )?
                      '(' 'evolve' term+ ')' ')'
trajId      ::=  id
```

2.6.3 Static Semantics

Each variable in an `evolvePred` must be an automaton parameter or a state variable.

Each `term` in an `evolvePred` must have type **Real** or **AugmentedReal**.

The `lvalue` in an `evolveValue` must have type **Real**.

[[Steve: Currently, values of type Int must be cast explicitly to values of type Real using the operator int2real, which has been added to the trait defining the Real datatype. Consider changing the Real datatype, as well as the type-checking rules for constructs such as assignments, to allow mixing elements of type Nat, Int, or Real, e.g., by defining operators such as __+__:Int,Real-¿Real.]]

[[Steve: The current syntax defines an `evolvePred` to be a `term`, and static semantic checks ensure that this `term` has the proper form. This was done to avoid making the grammar either ambiguous or overly complicated. Investigate changing the grammar.]]

[[Steve: Allow an `evolvePred` to have the form $a < b < c$, so that users are not forced to write two `evolvePreds` (i.e., $a < b; b < c$).]]

[[Steve: Change the grammar for terms to use the standard precedence for arithmetic operators, so that users can write $2 * a + b < c$ instead of $((2 * a) + b) < c$, as is required in LSL and IOA.]]

2.6.4 Semantics

If no trajectories are specified for an automaton, then the values of its state variables are constrained only by their dynamic types. If all variables in such an automaton are discrete, then the automaton is timing-independent; i.e., its trajectories are constant-valued functions over left-closed time intervals with left endpoint 0.

2.6.5 Conditions on algebraic expressions

If an `evolveValue` of the form **`d(v)`** appears in an `evolvePred`, then the `term` in that `evolvePred` must be integrable.

[[**Steve: The front-end will perform a static check for integrability if it can. Otherwise, it will generate a proof obligation—ideally, one that could be checked by a tool such as Macsyma.**]]

2.6.6 Conditions on invariants

The invariants (`invariant`) in trajectory definitions must be mutually exclusive. (Mutual exclusivity is needed to preserve prefix suffix and concatenation closure [5, 6].)

[[**Steve: The front-end will generate proof obligations that express these conditions.**]]

2.7 Trajectories

[[**Steve: We need to say what a trajectory is (domain, range, limit time).**]]

The trajectory definitions and urgency predicates collectively determine the set of trajectories for an automaton.

Notation: Suppose that τ is a fixed trajectory over some set of variables V and $v \in V$. We use the variable name v to denote the function that gives the value of v at all times during trajectory τ.

Given an urgency predicate and precondition for each locally contolled action, a trajectory definition a with invariant $inv(a)$, stopping predicate $stop(a)$, and a set of differential and algebraic equations $daes(a)$, defines a set of trajectories, say $traj(a)$. A trajectory τ belongs to $traj(a)$ if:

- For each $t \in dom(\tau)$, $\tau(t) \in inv(a)$,
- If $(t \in dom(\tau)) \wedge (\tau(t) \in stop(a) \vee (\exists b.\tau(t) \in urgent(b) \wedge pre(b)))$, then t is the limit time of τ,
- τ satisfies the set of differential and algebraic equations in $daes(a)$, and
- For each v of a simple type other than **Real**, $v(t) = v(0)$ for each $t \in dom(\tau)$. (For each v of one of the compound types listed in Section 2.4.1, we can determine a set of constituents $\{v_1, \ldots v_n\}$ of a simple type by recursing on the structure of v. We require that the above condition holds for each v_i for $1 \leq i \leq n$.)

If $inv(a)$ and $stop(a)$ are omitted from the specification, then they are assumed to be $true$ and $false$, respectively. A trajectory τ satisfies $daes(a)$ if at all times in $dom(\tau)$, all the locally controlled variables of type real satisfy the differential and algebraic equations (or inequalities) in $daes(a)$ with $\tau(0)$ defining the initial valuations.

Let $\mathcal{A}$ be an automaton and $TN_{\mathcal{A}}$ be the set of names of trajectory definitions of $\mathcal{A}$. Then the set of trajectories $\mathcal{T}_A$ for $\mathcal{A}$ is defined as the concatenation closure of the functions in $\bigcup_{a \in TN} traj(a)$.

Figure 2.1 summarizes the rules which determine if a trajectory τ satisfies some given algebraic and differential equations or inequalities. In the figure v is a locally controlled variable of type real, e is a real valued algebraic expression, and $\tau.e(t)$ is a function with domain $dom(\tau)$ that gives the value of e at t. Note that the figure does not cover all the possible inequalities one can form using the given syntax. The rest of the possible inequalities can be interpreted in the obvious way, using the same idea.

Equation	Interpretation
$v = e$	$\forall t \in dom(\tau),\ (\tau \downarrow v)(t) = \tau.e(t)$
$e \geq v$	$\forall t \in dom(\tau), \tau.e(t) \geq (\tau \downarrow v)(t)$
$e \leq v \leq e'$	$\forall t \in dom(\tau),\ \tau.e(t) \leq (\tau \downarrow v)(t) \leq \tau.e'(t)$
$d(v) = e$	$\forall t \in dom(\tau),\ (\tau \downarrow v)(t) = (\tau \downarrow v)(0) + \int_0^t \tau.e(t')\ dt'$
$e > d(v)$	$\forall t \in dom(\tau),\ (\tau \downarrow v)(0) + \int_0^t \tau.e(t')\ dt' > (\tau \downarrow v)(t)$
$e \leq d(v) \leq e'$	$\forall t_1, t_2 \in dom(\tau),\ \int_{t1}^{t2} \tau.e(t')dt' \leq (\tau \downarrow v)(t_2) - (\tau \downarrow v)(t_1) \leq \int_{t_1}^{t_2} \tau.e'(t')dt'.$

Figure 2.1: Interpretation of differential and algebraic equations and inequalities

We do not impose any conditions for solvability or well-definedness. The set of differential and algebraic equations may not be complete or some of the real variables may not be constrained at all.

Example 2.5

The automaton `A(u1,u2:Real)` has a single trajectory definition that states that time cannot progress beyond the point where `now` reaches value `u2` and that `now` evolves at the same rate as real-time. Supposing that `u1 <u2`, this automaton can do `a` within `u1` time units of the start and can keep doing `a` within `u1` time units of `a`'s last occurrence as long as `now <u2`. The urgency predicate for `a` enforces that either time stops or `a` is performed when `now` reaches the current deadline for `a`. Time cannot go beyond the point when `now` reaches `u2`. The only action that is enabled at that point is `end`, which sets `continue` to false and hence disables both of the discrete actions.

```
automaton A(u1, u2: Real)
  signature internal a, end
  states
    continue: bool := true,
    now: Real := 0,
    deadlinea: DiscreteReal := u1
  transitions
    internal a
        pre continue ∧ now ≠ u2
        urgent when now = deadlinea
        eff
          deadlinea := deadlinea + u1
    internal end
      pre continue ∧ now = u2
      eff
        continue := false
  trajectories
      trajdef always
        stop when now = u2
        evolve d(now) = 1
```

■

Example 2.6 The automaton `T(u,l,K,h:Real)` below specifies a thermostat [1]. Since the trajectories specify the continuous evolution of the temperature rather than timing related components, it would be more appropriate to call this a hybrid automaton rather than a timed automaton. Our goal in using this example is to illustrate that the syntax for algebraic and differential equations in TIOA is general enough to express non-trivial continuous evolution for state variables.

```
vocabulary locType
  type locType enumeration [on, off]

automaton T(u:Real, l:Real, K:Real, h:Real)
  imports locType
  signature
    internal on, off
  states
    x: Real := u
    loc:locType := on
  transitions
    internal on
      pre x = l and loc = off
      eff loc = on
    internal off
      pre x = u and loc = on
      eff loc = off
  trajectories
    trajdef heaterOn
      invariant x ≤ u and loc = on
      stop when x = u
      evolve d(x) = K(h-x)
    trajdef heaterOff
      invariant x ≥ l and loc = off
      stop when x = l
      evolve d(x) = -Kx
```

The temperature x, is governed by differential equations. When the heater is on the temperature follows the function $x(t) = \tau(0).xe^{-Kt} + h(1 - e^{-Kt})$, when the heater is off the temperature follows: $x(t) = \tau(0).xe^{-Kt}$. ■

2.8 Simulator

The set of NDR statements for the simulator is extended by a new **follow** statement, for use in simulating the effects of trajectories.

2.8.1 Surface Syntax

```
NDRStatement  ::=     assignment | NDRConditional | NDRWhile
                   |  NDRFire | NDRFollow
NDRFollow     ::=     'follow' IDENTIFER 'for' term
```

2.8.2 Intermediate Language Syntax

```
statement  ::=       assignment | conditional | loop
                |    simfire | ndrfire | ndrfollow | ndrwhile
ndrfollow  ::=  '('  follow trajectoryId term ')'
```

2.8.3 Static Semantics

The identifier in a **follow** statement must be the name of a trajectory defined in the automaton.

Each variable in the term in a **follow** statement must be an automaton parameter or a state variable.

The type of the term in a **follow** statement must be **Real**.

[[**Steve: Still to come – notations for paired simulations.**]]

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicolin, A. Olivero, J. Sifakis, and Yovine S. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[2] S. Garland, N. Lynch, Joshua Tauber, and M. Vaziri. *IOA User Guide and Reference Manual.* MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2003. Available at `http://theory.lcs.mit.edu/tds/ioa.html`.

[3] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917, MIT Laboratory for Computer Science, 2003. Available at `http://theory.lcs.mit.edu/tds/reflist.html`.

[4] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. Technical report, Cancun, Mexico, 2003. Full version available as Technical Report MIT/LCS/TR-917.

[5] S. Mitra, Y. Wang, N. Lynch, and E. Feron. Safety verification of pitch controller for model helicopter. In O. Maler and A. Pnueli, editors, *Proc. of Hybrid Systems: Computation and*

Control, volume 2623 of *Lecture Notes in Computer Science*, pages 343–358, Prague, the Czech Republic April 3-5, 2003.

[6] Sayan Mitra. HIOA+: Specification language and proof tools for hybrid systems. Unpublished manuscript.