Recent Fixes (v0.2.2) 3
  *v0.2.2 - 2/7/2008 3
  v0.2.1 - 1/15/2008 3
  v0.2.0 - 10/25/2007 3
  v0.1.9 - 7/18/2007 3

Known Issues (v0.2.2) 4
  General 4
  Tempo Simulator 4
  Graphical User Interface 4

Language Changes (v0.2.2) 5
  *Quantifiers on Sets 5
  *Nested Transitions 5
  Nested Trajectories 5
  Parametric Trajectories 6
  Empty Trajectories 6
  *Include 6
  Built-in Operators 6
  Automaton State Initialization 7
  Automaton State Access 7
  Primed Variables 7
  Transition Variable Inference 7
Transition Relations 8
*Trajectory Relations 8
Simulation Relations 8
Type Cast 9
Transition Locals 9
Terminators and Separators 9
Empty Automata 10
Let Functions 10
Type Shorthands 11
Component Arrays 12
Set Constructor 12
For Statements 12
Through Notation 13
String Literals 13

Plugins Notes (v0.2.2) 13
1. PVS 13
2. *UPPAAL 16
3. *Simulator 17
4. LaTeX Translator 22
Recent Fixes (v0.2.2)

* v0.2.2 - 2/7/2008
- Subsections of this document which have been updated in v0.2.2 have been marked with an asterisk (*).
- A new version of the model checker is included with this release. Please see the model checker manual for details.
- A new syntax for quantifiers has been added to support quantification over sets.
- The syntax of follow, fire, and simulation proof statements has been updated to better support nest composite automata.
- Several bugs were fixed relating to how paired output and input actions were simulated in composite automata.

v0.2.1 - 1/15/2008
- An updated draft of the Tempo language guide is included with this release.

Tempo Simulator
- The syntax of stop when clauses and choose where clauses has changed, please see the simulator plugin section of this document for details.
- Trajectories with parameters are now supported.
- The supported stop when expressions have been increased.
- All forms of expressions are supported in simulation relations.
- Parametric automata are supported in simulation relations.
- Choose statements have been revised.

v0.2.0 - 10/25/2007
- A new draft of the Tempo language guide is included with this release.
- More changes were made to the Tempo syntax, see the Language Changes section of this document for details.
- Many updates were made to the PVS plugin, see the PVS Plugin Notes for more details.
- A ‘seed’ command line option was added to the Tempo simulator so the results of random choices could be fixed for multiple executions.

v0.1.9 - 7/18/2007
- More changes were made to the Tempo syntax, see the Language Changes section of this document for details.
• The simulation relation using statement is now checked correctly.

• Char is now correctly identified as a finite type and can be used with loops and quantifiers in the simulator

• The graphical user interface now supports files with tabulation in them.

• The UI guide has been rewritten and updated to the latest version of the tool.

• DiscreteReal is now correctly interpreted as a subtype of Real.

• Implicit importing of the List vocabulary in simulation relations was fixed.

• The front end's algorithms for type inference in quantified expressions were enhanced.

**Tempo Simulator**

• Basic simulation proofs are now supported

• Nested for loops and quantifiers now simulate correctly.

• An equality test bug related to enumerated types has been fixed.

• Signature where clauses are now evaluated

• Fire output statements now output their values to the given parameters

• Invariant failure messages were enhanced to be more descriptive

**Known Issues (v0.2.2)**

**General**

• The automatic invariant validation mode of the new PVS integration is not yet implemented.

• Initial tests indicate the Tempo toolset is compatible with Windows Vista, but it is not extensively tested under Windows Vista. If you experience any problems while using the Tempo toolset in Windows Vista please contact support.

**Tempo Simulator**

• Break points inside simulation relations may behave incorrectly. Single stepping through simulation relations is recommended.

• If a node in the variables view is closed it cannot be reopened

**Graphical User Interface**

• (OS X Only) The output file generated by the UPPAAL plugin is placed in the Tempo User Interface installation directory with the name ‘tempo.xta’. The same location as the tempo.app file. For now, on OS X, the command-line interface is best to produce file output of the UPPAAL plugin. (RT 107)
**Language Changes (v0.2.2)**

**Quantifiers on Sets**
Quantifiers on sets have been added to Tempo and can be specified as follows,

\[(\forall | \exists) \ <variable\ name> \ in \ <set\ expression> \ <predicate\ expression>\]

Some examples,

\[\exists y \ in \ s \ y = 5\]
\[\forall y \ in \ s \ y < 10\]
\[\forall y \ in \ -3..3 \ y > -4 \ \&\ y < 4\]
\[\exists y \ in \ {-8,-6,-4} \ x > -7 \ \&\ x < -5\]

**Nested Transitions**
Consider the following nested composite automaton,

```plaintext
automaton AutoA
signature output out
states x:Int := 10;
transitions output out
...
automaton AutoC
components
A1:AutoA;
A2:AutoA;
...
automaton AutoD
components
C1:AutoC
C2:AutoC
```

Previously a fire statement in the schedule of 'AutoD' would appear as,

```plaintext
fire output C1.out;
```

Now these transition references must be fully dereferenced as follows,

```plaintext
fire output C1.A1.out;
```

Similar syntax changes have been extended to simulation relations as well.

**Nested Trajectories**
Trajectories of a nested composite automaton can now be specified as follows,

```plaintext
automaton AutoA
...
trajectories
trajdef T1
...
automaton C
components
A1:AutoA;
A2:AutoA;
...
automaton D
components
  C1:C
  C2:C
schedule do
end

**Parametric Trajectories**

Parametric trajectories have been added to Tempo. A parametric trajectory can be specified and referenced as follows,

automaton A
  states t:Real;
  trajectories
    trajdef T(i:Int)
      evolve
        d(t) = i;
end

schedule do
  follow T1(2) duration 10;
end

**Empty Trajectories**

Trajectories are no longer required to have evolve clauses. An empty trajectory can be specified as follows,

automaton A
  states t:Real;
  trajectories
    trajdef T1

**Include**

An include command has been added to Tempo and can be specified as follows,

include "<file path>"

The file path is a relative path from where the JVM was executed.

**Built-in Operators**

The Set, Mset, and Seq vocabularies were updated to support a `\notin` operator which is the complement of the `\in` operator. The `\notin` operator can be used as follows,

automaton A
  states
    s:Set[Nat];
    x:Nat;
    ...
  if(x \notin s) ... fi
The Real and AugmentedReal vocabularies were updated to support a square root operator, 'sqrt', which can be used as follows,

```plaintext
automaton A
    states r:Real;
    ...
    r := sqrt(r);
```

**Automaton State Initialization**

Automaton state variables can no longer be used to initialize other state variables. States block like this are no longer accepted,

```plaintext
states
    x:Nat := 5;
    y:Nat := x*2;
```

**Automaton State Access**

Automaton state variables cannot be written to from schedule blocks. State variables can be read from but cannot appear on the left hand side of an assignment statement. For example, the following state assignment is no longer accepted,

```plaintext
states
    x:Nat;
    ...
    schedule do
        x := 5;
    od
```

**Primed Variables**

Primed variables are no longer supported in effects clauses. They can only appear in ensuring clauses as follows,

```plaintext
states
    a:Nat;
    b:Nat;
    c:Nat;
    ...
    eff
        a := choose;
        b := choose;
        c := a + b;
        ensuring a' > b';
```

**Transition Variable Inference**

Transitions argument expressions have been restricted further and now can only be an identifier or a literal value. If an identifier is found, it is interpreted as the desired name for that transition's parameter. For example,

```plaintext
automaton AutoA
    ...
    states
```
transitions
  input foo(x)   %ok
  input foo(10)  %ok
  input foo(10+x) %not ok, expressions are no longer supported
  input foo(a+x) %not ok
  input foo(a)   %ok, there is only one identifier ‘a’,
                %this ‘a’ will shadow the state variable ‘a’
...

Transition Relations
Transition relations no longer require the types of their parameters to be specified. Only a parameter name is required. The type of that parameter is inferred from the automaton's signature. Previously a transition relation would be written like this,

proof
  for input foo(a: Nat) do
    fire input bar(a+5);
  od

The same transition relation is now written as follows,

proof
  for input foo(a) do
    fire input bar(a+5);
  od

*Trajectory Relations
Trajectory relations now require that only an identifier is specified in the duration value. For example,

proof do
  for trajectory T1 duration x ignore %ok, there is exactly one identifier ‘x’
  for trajectory T2 duration x+5 ignore %not ok, a complex expression
  for trajectory T3 duration 5 ignore %not ok, no identifier
  od

Simulation Relations
Simulation relations have been updated to have names and parameters. Previously a simulation relation would be written like this,

forward simulation from A(x, y) to B(x, y) :
  A.i = B.i \ A.j = B.j;
...
end

The same simulation relation is now written as follows,

forward simulation F(x, y: Int) where x <= y
  from A(x, y)
  to B(x, y)
  mapping
    A.i = B.i \ A.j = B.j;
This defines a simulation relation named F with integer parameters x and y. The where clauses is used to specify restrictions on the parameters. The ‘:’ marking the start of the relation between the two automata has been changed to the keyword ‘mapping’.

**Type Cast**

A syntax for casting has been added to the Tempo Language. Any expression can be cast to another type by preceding the expression with a type name wrapped in parenthesis. For example, the follow code will cast the Augmented Real x into a Real,

```plaintext
states
 x:AugmentedReal
...
(Real)x
```

**Transition Locals**

The syntax for transition locals has been updated to be consistent with other language constructions. Previously locals were specified in the transition signature as follows,

```plaintext
output out(x,y;local i:Int, j:Int)
```

Now transition local variables are specified with a locals block that follows directly after the transition’s signature. Additionally locals have been updated to support initial values. For example,

```plaintext
output out(x,y)
locals
 i:Int;
 j:Int := x * y / 2;
...
```

**Terminators and Separators**

Many constructions have been “sanitized” to have a consistent use of comma and semicolon. Tempo now has the convention that comma is used to separate elements in a list and semicolon is used to terminate elements in a list. A brief breakdown of which lists are separated and which are terminated,

**Semicolon terminated**

- Automaton States
- Automaton Components
- Let Functions
- Invariants
- Preconditions
• Evolve, Stop When, Initially, Ensuring, and Urgent When Clauses

• Statement Blocks

**Comma separated**

• Type Declarations

• Vocabulary Operators

• Vocabulary Imports

• arguments and formals of functions, automata, and actions

Additionally to help make statement blocks more natural the statement terminator can be omitted when closing a while or if statement with a ‘od’ or ‘fi’. Previously a statement block would specified as follows,

```plaintext
while(y < 5) do
  if(z > 0)
    z++;
  else
    x++;
  fi;
od;
```

can now be written more naturally as,

```plaintext
while(y < 5) do
  if(z > 0)
    z++;
  else
    x++;
  fi
od
```

**Empty Automata**

The smallest Tempo specification one can write is now two lines long. Namely, the following

```plaintext
automaton A
states
```

is a correct specification.

**Let Functions**

The syntax of functions is now more consistent and closer to operator definitions in vocabularies. Functions were specified as follows

```plaintext
let x = 3.14;
let y(i:Int) = i * 2;
let z(j:Real, k:Real) = j + k;
```
The new syntax follows the name : type convention used in so many other constructions. The above examples can now be written as

```plaintext
let x() : -> Real = 3.14;
let y(i) : Int -> Int = i * 2;
let z(j,k) : Real,Real -> Real = j + k;
```

The first line defines a constant function with no argument. The second is a function from an integer to an integer where the name of the argument is i. The third function takes two arguments j and k both of type Real and produces a result of type Real. The syntax has also been relaxed to support recursive functions: For instance, the well-known factorial function can be written as,

```plaintext
let fact(x) : Int -> Int =  
  if x = 1
    then x
  else x * fact(x-1);
```

We currently do not support mutually recursive definitions. Note that recursive functions are now supported in the front-end and the new simulator.

**Type Shorthands**

The syntax of the type shorthands (Tuple, Enumeration, Union) has been changed to be more uniform with other type declarations. Previously the type shorthands were specified in vocabularies like this,

```plaintext
vocabulary Shortcuts
  types
    tup   tuple [i: Int, j: Int],
    enum  enumeration [x, y, z],
    uni   union [ff: Int, gg: Real]
end
```

These were changed to be consistent with all other type declarations which are of the form 'name : type'. The example should thus be rewritten into

```plaintext
vocabulary Shortcuts
  types
    tup   : Tuple [i: Int, j: Int],
    enum  : Enumeration [x, y, z],
    uni   : Union [ff: Int, gg: Real]
end
```

Note, the addition of the colon and the capitalization of the keyword. Additionally these types can be specified directly and do not require a global type declaration or an encapsulating vocabulary. For example,

```plaintext
automaton A
  states
    colors : Set[Enumeration[red, green, blue]];
    pair   : Tuple [p:Int, s:Int];
    numList : Seq[Union[i:Int, r:Real]];
```
**Component Arrays**

The predicate attached to parametric component definition are now more permissive. The sole remaining requirement is that a variable of a non-finite type appears at least once in the where clause. For example

```plaintext
Types
  Color : Enumeration[red, green, blue]
...
components
  C1[j:Nat]: AutoA where j < 10;
  C2[j:Color]: AutoA;
  C3[i:Nat, j:Color, k:Nat]: AutoA where i < 3 \& k < 3 \& j ~= green;
```

The first line specifies C1 as an array of automatons AutoA and there is one entry for each natural number j less than 10. The second line defines C2 as an array of AutoA with one instance for each value j that belongs to type Color. The third line states that C3 is a three-dimensional matrix of AutoA automaton but restricts i and k to be in [0..2], j in the set of all colors (but green).

**Set Constructor**

A new set constructor has been added. It takes a finite type and a predicate and generates the set of all elements of that type which satisfy the predicate. For example,

```plaintext
Types
  Color : Enumeration[red, green, blue]
...
x := {c:Color where true};
y := {c:Color where c ~= red};
z := {c:Color where c = green};
```

yields the following sets,

- x -> {red, green, blue}
- y -> {green, blue}
- z -> {green}

**For Statements**

The semantic checking of a for loop which uses the ‘in’ notation (ie for x:Int in myIntSet) has been updated so that the item provided after the ‘in’ must be of type, Set[T], Mset[T], or Enum. Also the type of the for loop variable is now optional for the for loop-in. Some examples of the common and new for loop syntax include,

```plaintext
for y:Nat where x < 10 do ... od
for y in mySet do ... od
for y in myMset do ... od
for y in {0,1,2,3} do ... od
for y in Color where x < 10 do ... od
```
**Through Notation**

The ‘..' operator has been added to the Set and Mset vocabularies as a shorthand for generating lists of consecutive values. For example,

```plaintext
s: Set := -3..3
```

yields the following set,

```plaintext
{-3,-2,-1,0,1,2,3}
```

This notation can be used in conjunction with for loops as follows,

```plaintext
for y in -3..3 do
   sum := sum + y;
od
```

**String Literals**

String literals have been added to Tempo. They can be used as follows,

```plaintext
s: String := "HelloWorld"
```

the exact lexical specification is,

```plaintext
"(\'a\'..'z'|\'A\'..'Z'|\'0\'..'9\')*"
```

The acceptable characters for a string literal is defined by the character vocabulary.

---

**Plugins Notes (v0.2.2)**

Each plugin imposes its own set of restrictions on the core Tempo language. Depending on which tool is selected, your model may or may not comply and the front-end will report additional errors that are tool dependent. The remainder of this document briefly reviews each tool and the restrictions imposed by that tool. Complete documentation on how to use each plugin can found in the documentation directory.

1. **PVS**

   **PVS Configuration File**

   A configuration file is no longer required for Tempo to PVS translation. The Tempo UI has been enhanced to support the PVS code generation options from the PVS plugin preference page. An editor has also been added to the Tempo UI to help modification and maintenance of configuration file. To take advantage of the new configuration editor the configuration file must end with the ‘.cfg’ extension. When working from the command line, code generation options can be specified in the configuration file or as command line arguments.
**PVS Integration.**  
The PVS Translator plugin now supports an integration with the PVS 3.2 application. After translation is complete, the plugin can automatically start PVS and begin proving lemmas required by the specification. Due to the limitations of PVS 3.2 this integration is limited to Linux. For more details on setting up this integration, please see the PVS manual supplement included in the docs folder.

**While loops**  
While loops are not supported by the PVS translator. If a loop is required for loops are supported by the translator.

**Parametric Types**  
Parametric types in vocabularies and automaton specifications are not supported. For example, the following are not supported:

- `vocabulary MyVoc defines MyType[T] ...`
- `vocabulary myVoc(T: type) ...`
- `automaton test(mytype:type) ...`

A work-around is to declare a type construct within a vocabulary, and then use the “inc1” option to specify a PVS uninterpreted type in an include file. For example, one could write the following:

```plaintext
vocabulary myvocab  
  types mytype  
end  
imports myvocab  
automaton test  
  signature output out  
  states x:mytype, y:Seq[mytype], z:Null[mytype] ...
```

Then, one should use the “inc1” option to include a file containing an uninterpreted type in the PVS output. For example, an include file named “file.inc” contains:

```plaintext
mytype: TYPE
```

Then the configuration file should contain the option “inc1:file.inc” on a single line.

**Built-in Types**  
All built-in types are supported except “String” and “Mset”. In addition, for the “Seq” type, the assignment operator is not supported. Thus, the following statement is not supported, where “s” is of type “Seq”:

```plaintext
s[i] := x
```
Identifiers
Reserved words in PVS should not be used as identifiers in the Tempo specification. In addition, the translator also uses a fixed list of names in the output of the translation, and these names should be avoided whenever possible to prevent unnecessary overloading in PVS. For example, names such as “actions”, “delta_t”, “time”, “theory”, “begin”, etc. should not be used.

Action and transition signatures
Formal parameters of an action or transition should not use the “const” keyword, and should not be literals. For example, the following is not allowed:

\[
\text{input send(const i, const j)}
\]

One could rewrite the above using a “where” clause into an acceptable form:

\[
\text{input send(i1:Int, j1:Int) where i=i1 /\ j=j1}
\]

As another example, the following is not allowed:

\[
\text{output out(0)}
\]

Again, it is possible rewrite using a “where” clause to obtain an acceptable form:

\[
\text{output out(i) where i=0}
\]

Action and transition constructs
The following constructs within actions and transitions are not allowed:

- choose
- ensuring
- hidden
- local

Trajectory evolve clause
Evolve clause of a trajectory should be either a constant differential equation or a constant differential inclusion. Higher orders not supported currently. For example, the following expressions are allowed:

\[
\text{d(x) = k, d(x) >= k, d(x)<= k, d(x) > k, d(x) < k,}
\]

where “k” is a literal constant.

Simulation relations
Only forward simulations are allowed. Backward simulations are not supported.

When a simulation relation is defined from automaton A to automaton B, both A and B should have the same set of external actions.

Proof entries in a simulation relation are ignored.
Schedules
Schedule blocks are ignored.

2. *UPPAAL

*General
Many changes have been made to the uppaal translation strategy. Please review the uppaal translator’s manual for details.

Variable types
All the declared state variables could be only Int, Nat, Bool, Enum and the array of these four types except time variable. Time variable can be only Real, which means set, map, tuple, sequence, union etc. are all not allowed, imported vocabulary can only be Enum. Also, “type” and “local” keywords are not allowed.

Disallowed syntactic constructions
1. Where clause are not allowed, e.g., automaton header, signature etc. with one exception, where clauses are allowed when defining for loops
2. “let” clause is not allowed
3. “choose” clause and “initially” clause are not allowed
4. “urgent when” clause is not allowed
5. “ensuring” clause is not allowed
6. “hidden” actions are not allowed
7. Universal and existential quantifiers are not allowed (\(\forall A \text{ and } \exists E\))
8. Automaton state dereference is limited to components of composite automata
9. The “\(\infty\)” constant is not allowed
10. Generally if-then, if-then-else, for loop, and assignment statements allowed, with a special exception for input transitions, please see the Transitions section.

Constructions that are discarded (The translator will simply discard the construction and carry on with the translation):
1. Task blocks are ignored.
2. Invariant statements are ignored.
3. Schedule block are ignored.
4. Simulation (forward simulation, backward simulation) blocks are ignored

Functions
Only the following functions are supported by the model checker,

\[\text{div, mod, pred, succ, min, max}\]
**Arrays**

Arrays are supported by the model checker, but there are restrictions on the types which are used to define the array. Specifically, the domain of the array can only contain the types, Enum, Nat, and Int, and the co-domain of the array must be of the type, Enum, Bool, Nat, or Int.

**Transitions**

1. Internal transitions can't have parameters

2. Precondition clauses cannot contain disjunctions

3. In the effects clause of an input transition, if-then-else statements cannot be used. If an if-then statement is used, then it must be the first statement and all other statements must be contained in its then clause.

**Trajectories**

All basic automata must define at least one Trajectory.

Trajectory evolve clauses are limited to $d(t)=1$ (time evolves at constant rate 1). Trajectories have the format

```
stop when time = ...
```

The differential equation must have the form

```
d(t)=c
```

where $c$ is a constant value typed as a real. The variable $t$ in $d(t)$ must be a real.

3. *Simulator*

The following restrictions on the core Tempo language are for the new Tempo simulator. If you would like the language restrictions for the old tempo simulator, please visit the release archives at [www.VeroModo.com/tempo](http://www.VeroModo.com/tempo). The restrictions that are listed below will be progressively removed with new releases.

**Simulation Proofs**

Simulation proofs of basic automata are now supported by the tempo simulator. Currently composite automata are not yet supported in simulation relations. This is a new feature and we are working to increase it's language coverage.

**Fire Output**

Fire output statements now correctly implement their in/out behavior. Once a fire output instruction has completed, it will write it's outputs back into the variables used as inputs. For example,

```
transitions
  output foo(x)
  eff
    x := x*10;
...
```
schedule
states
    a: Int := 5;
    print a; %prints ‘5’
    fire output foo(a); % 5 is passed to foo(x), then 50 is written to a
    print a; %prints ‘50’

Because the formals (‘a’ in the example above) must be written to, the expression must be a valid left hand value or a literal value as follows,

    10 %a literal value
    x %an identifier
    x.a %a tuple field
    x[i] %a array field (note: ‘i’ can be any arbitrary expression)

An automaton state variable may be used as a fire output formal, but an additional restriction is enforced, that variable cannot be modified during the execution of the output action.

**Primed Variables**
The simulator now supports primed variables in ensuring statements. The primed operator only effects automaton state variables. Access to all other variables is identical with our with out a primed variable.

**Composition**
Automata composition is allowed with some restrictions.

• Components cannot be initialized as arrays. For example,

    components
        comp4[x: Int]: A where x < 100 \&\& x > 10;

    is not supported.

• Components must be given names. For example,

    components
        A; B;

    is not supported.

**Quantifiers**
Arbitrary quantified expressions in the simulator can only be defined on enumerable types, Bool, Nat, and user-defined Enumerations. For example,

    Types
        Color : Enumeration[red, green, blue]

    \E b: Bool b = true
    \A c: Color c = green
    \E n: Nat n > 100

When working with quantifiers over the Nat type keep in mind it is easy to create a test which never terminates. For example,
\E n:\text{Nat} \ n \ < \ 0 \\
\A n:\text{Nat} \ n \ > \ 0

In this case the simulator will execute indefinitely in an attempt to test all natural numbers. This often results in a JVM out of memory error. Note that quantifiers over sets are fully supported by the simulator.

**Set .. Operator**
The “..” set constructor is only supported for operands of type Int, Nat, and Bool. For example,

\[
x : \text{Set(Int)} := -5..5
\]

Yields the follow set,

\[
x \rightarrow \{-5,-4,-3,-2,-1,0,1,2,3,4,5\}
\]

*Choose statements*
Choose statements may appear in the state declarations and in the body of the transition. The choose statements have to be defined on numerical built-in data types (i.e. Bool, Nat, Int, Real, DiscreteReal, AugmentedReal) and maybe of the following form:

\[
:= \text{choose} \\
:= \text{choose } x \text{ where } <\text{constraint expr}> \\
:= \text{choose } x \text{ where } <\text{constraint expr}> \ \land <\text{constraint expr}>
\]

where the `<constraint expr>` may be a simple relational operator (>, >=, <, <=) with parameter x and an expression not containing x. The choose variable, x, must be on the left hand side of the constraint expression. If a conjunction is used to specify a range the lower bound must be on the left hand side and the upper bound on the right hand side.

*Stop when statements*
The simulator currently restricts stop when statements to the following EBNF grammar,

\[
\text{root:} \\
| \text{conjunct} \\
| \text{recursiveRoot} \\

\text{conjunct:} \\
| <\text{expr} \ \land \ \text{recursiveRoot} \\

\text{recursiveRoot:} \\
| <\text{expr} \\
| <\text{constraint} \\
| <\text{disjunct} \\

\text{disjunct:} \\
| \text{recursiveRoot} \ \lor \ \text{recursiveRoot} \\

\text{constraint:} \\
| '\text{evolve variable}' \ ( = | <= | >= ) \ <\text{expr}\
\]

19
An *expr* is any expression that does not contain an evolve variable. Some common stop when statements,

```plaintext
evolve  
d(x) = 1;  
d(y) = 1;  
d(z) = 1;  
...  
stop when x >= 10  
stop when y = 3 \ / z <= 5  
stop when \ -\failed \ / \ x = stopTime
```

The greater than constraint is only supported with positive evolve rates. The less than constraint is only supported with negative evolve rates. The equality constraint is supported with any rate.

*Evolve statements*

The simulator currently restricts the evolve statements to the following formats,

```plaintext
d(x) = <constant expression>;  
d(x\_clock) = <constant expression>;  
```

where a <constant expression> does not contain an evolve variable. All other evolve statements are not supported, some examples of these are:

```plaintext
d(x) > 5;  
d(y) = x;
```

*Schedules*

The simulator must have a schedule block in order to simulate any specification. If there is only one schedule block the simulator will execute this block. If there is more than one automaton with a schedule block or the automaton with a schedule block has parameters a simulate block must be used to specify an automaton and instantiate it’s parameters.

*Det statements*

Choose det blocks are not supported by the Simulator.

*For-where statements*

Simulator supports only for-where statements that have as a less than predicate, for example:

```plaintext
for j:Nat where j < 10 do  
...  
od;
```

All forms of the for-in loop are supported by the simulator.

*Smart fire*

Currently the simulator requires the user to provide a schedule which includes fire statements that are followed by an action kind (aka input, output, or internal) and a name, with parameters if any are needed. Fire statements with out an action kind and name are ignored.
**Transition Formals**
The simulator only supports transition formals which are identifiers. An example of acceptable transition formals,

```
transition output out(a,b)
```

An example of unsupported transition arguments,

```
transition output out(a+5,b/2)
```

**Automata Formals**
The simulator does not support parametric automata with type parameters. For example,

```
automaton A(T:type)
  states
    x:T
  simulate do
    run A(Int);
  od
```

Parameters of other types are supported.

**User defined types**
User defined types are not supported by the simulator. The simulator supports all built in types and user defined type shorthands (Tuple, Enumeration, and Union).

**User defined operators in vocabulary**
User defined vocabulary operators are supported by the simulator if they only use the Tempo built in types. A native java implementation of the vocabulary must be provided to run a simulation with the user defined vocabulary. For more details on developing a native java implementation of a vocabulary see the simulator manual supplement included in the docs folder.

**Type Shorthands (Tuple, Enumeration, Union)**
The simulator supports shorthands of all types. For example,

```
vocabulary Shortcuts
  types
    tup   : Tuple [i: Int, j: Int],
    enum  : Enumeration [x, y, z],
    uni   : Union [ff: Int, gg: Real]
end

automaton A
  states
    colors  : Set[Enumeration[red, green, blue]];
    pair    : Tuple [p:Int, s:Int];
    numList : Seq[Union[i:Int, r:Real]];
```

**Ignored Constructions**
There are several constructions which the simulator will ignore. These are
• Task Blocks
• Smart Fire statements
• Composite Automaton Hidden blocks
• Simulation proof start blocks

4. **LaTeX Translator**
The LaTeX translator supports the complete Tempo language specification, but requires the TempoMacro file to render the output. The TempoMacro file will be generated automatically by the LaTeX Translator plugin if no macro file already exists in the output directory. If you wish to force an overwrite of the existing TempoMacro file, the “-makeMacro” command line option will force the existing macro file to be overwritten.

*Enjoy!*