TEMPO RELEASE NOTES

v0.2.3 (BETA)

6/27/2008

*v0.2.3 - 6/27/2008

- 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
- The Tempo GUI has been updated to the Eclipse RCP 3.3
- The latex translator now correctly supports includes
- The simulator will now terminate after the correct number of steps has been reached
- Equality operators in parametric vocabularies now work in the Tempo simulator
- Richer evolve variables are supported in the simulator

v0.2.2 - 2/7/2008

- 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.

Known Issues (v0.2.3)

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.

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 (v0.2.3)

*Quantifiers on Sets

The "where" keyword has been added to the syntax for quantifiers on sets as follows,

```
(\A | \E) <variable name> in <set expression> where <predicate expression>
```

Some examples,

```
\E y in s where y = 5
\A y in s where y < 10
\A y in -3..3 where y > -4 /\ y < 4
\E y in {-8,-6,-4} where x > -7 /\ x < -5
```

*Include

The include command has been enhanced to support lists of elements and identifiers as follows,

```
include “<file path>”
include <id>, “<file path>”
include “<file path>”, “<file path>”, “<file path>”
```

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

Simulate Locals

Local variables have been added to simulate blocks as follows,

```
simulate
locals
  names:Set[String] := {“one”, “two”, “three”};
do
  for name in name do
    run A(name);
    ...
  od
od
```

Nested Transitions

Consider the following nested composite automaton,

```
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,

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

Now these transition references must be fully dereferenced as follows,

```
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,

```
automaton AutoA
...
trajectories
  trajdef T1
```

```
...
automaton C
  components
    A1:AutoA;
    A2:AutoA;
...
automaton D
  components
    C1:C
    C2:C
  schedule do
    follow C1.A1.T1, C1.A2.T1, C2.A1.T1, C2.A2.T1 duration 10;
  od
```

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;

schedule do
  follow T1(2) duration 10;
od
```

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
```

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,

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

Automaton State Initialization

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

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

Automaton State Access

Automaton state variables can no longer 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,

```
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,

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

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;
...
end
```

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,

```
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,

```
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,

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

Empty Automata

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

```
automaton A
states
```

is a correct specification.

Plugins Notes (v0.2.3)

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 compete, 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:

```
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:

```
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":

```
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:

```
input send(const i, const j)
```

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

```
input send(i1:Int, j1:Int) where i=i1 /\ j=j1
```

As another example, the following is not allowed:

```
output out(0)
```

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

```
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:

```
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. Section 2 of the uppaal translator manual contains the complete list of language restrictions.

Variable types

The following types are supported, Int, Nat, Bool, Real, Enumeration, Array, Set, Seq, and Tuple. Type generics 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. Automaton state dereference is limited to components of composite automata
8. The "\infty" constant is not allowed

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

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 the release archives at 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:Nat n < 0
\A n: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 : Set[Int] := -5..5
```

Yields the follow set,

```
x -> {-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, DiscrteReal, AugmentedReal) and maybe of the following form:

```
:= choose
:= choose x where <constraint expr>
:= choose x where <constraint expr> /\ <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,

```
root:
  | conjunct
  | recursiveRoot

conjunct:
  | expr /\ recursiveRoot

recursiveRoot:
  | expr
  | constraint
  | disjunct

disjunct:
  | recursiveRoot \/ recursiveRoot

constraint:
  | 'evolve variable' ( = | <= | >= ) expr
```

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

```
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 grater 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,

```
d(<lvalue>) = <constant expression>;
```

where a <*constant expression*> does not contain an evolve variable and <*lvalue*> is a valid left hand value of an assignment (i.e. variable names, tuples, and arrays). All other evolve statements are not supported, some examples of these are:

```
d(x) > 5;  % '>' is not supported
d(y) = x;  % 'x' cannot appear in the expression because it is an evolve variable
```

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.

For-where statements

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

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

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

Transition Formals

The simulator only supports transition formals which are identifiers. An example of acceptable transition formals,

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

An examples 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
- Composite Automaton Hidden blocks
- Simulation proof start blocks

4. LaTeX Translator

The LaTeX translator supports the complete Tempo language specification, but requites 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!