TEMPO RELEASE NOTES

v0.1.4 (BETA)

12/14/2006

v0.1.4 - 12/14/2006

- Subsections of this document which have been updated in v0.1.4 have been marked with an asterisk (*).
- A new plugin has been added to the distribution. The tempo2tex plugin will translate a Tempo specification into a LaTeX document.
- A bug in the front end, which prevented checking of schedules with actions which have types as parameters was fixed. (RT 175)
- The set notation, {__}, has ben extended to support multiple elements. Please see the language changes section for more details.
- The simulator now supports follow statements in composite automata schedules.

v0.1.3 - 11/9/2006

- After a file is analyzed if errors were generated, the number of errors is displayed. (RT 137)
- There are new language changes including, an alias type shorthand, a new function in the Map vocabulary, and instantiation of automata for simulations. Please see the language changes section for more details.
- The restrictions on the simulator have been relaxed. Please review the updated restrictions in the plugin notes section. Updated points include composite automata and choose statements.
- The restrictions on the model checker have been relaxed. Please review the updated restrictions in the plugin notes section. Updated points include statements in effect clauses.

v0.1.2 - 9/25/2006

- After a file is analyzed if errors were generated, the number of errors is displayed. (RT 137)
- Error messages produced by misuse of the type selector construction are less vague (RT 138)
- The text representation for tuple, union, and enum types used in error reporting has been simplified so complex types are more readable.
- Performance problems with composed shorthand types have been improved. More performance increases are on the way in future releases. (RT 140)

Known Issues (v0.1.4)

Graphical User Interface

- (OS X Only) The output files generated by the UPPAAL plugin are burried in the Tempo workspace inside the tempo.app file. For now, on OS X, the command-line interface is best to retrieve the file output of the UPPAAL plugin. (RT 107)
- Currently analyzing multiple Tempo specification files as a group is not supported in the Tempo Graphical User Interface. For this reason the PVS examples, timeout and TwoTaskRace cannot be fully translated with the Graphical User Interface. (RT 110)

Language changes (v0.1.4)

*Set Notation

The set notation, "{__}" has been upgraded to support lists for multiple elements. For example,

```
states
  x : Set[Nat] := {0,2,4,6,8}
```

Any curly brace construction in a vocabulary can support lists of multiple elements by using the type signature "List[T]". For example,

```
{__} : List[T] -> Set[T]
```

*end Keyword

To speed up parsing performance an "end" token is now required to complete Vocabulary and Simulation blocks. For example,

```
vocabulary Foo
  ...
end

forward simulation from A to B :
  ...
end
```

Alias Type Shorthand

The type shorthand, ":" can be used to alias a complex type to a single name, similar to a defType in C. For example,

```
types
  short : Array[Seq[Real], Nat]
```

The type "short" can now be used to represent a vector of sequences.

Map Remove Function

A "remove" function was added to the Map vocabulary. Remove takes a map and a key and returns a map with the value of the given key removed. An example of the syntax,

```
x:Map[Bool, Int]
x := remove(x, -5);

x:Map[Bool, Nat, Nat]
x := remove(x, 3, 7);
```

Simulation Automata Instantiation

Parametric automata used in simulation relations must be provided arguments for instantiation. For example, automata

```
automaton A(x:Int)
...
automata B
```

previously could be simulated,

```
forward simulation from A to B:
```

now a parameter must be provided to automaton A,

```
forward simulation from A(5) to B:
```

Typing

The new implementation considers that Bool is a subtype of Nat. The type hierarchy for numbers is thus as follows

```
Bool <: Nat <: Int <: Real <: AugmentedReal
```

Associativity and Precedence

The new implementation supports traditional associativity for all binary operators. In particular, an expression like

```
x + y < z * w
```

now yields the "natural" parse tree

```
((x + y) < (z * w))
```

Rather than

```
((((x) + y) < z) * w)
```

The precedence and associativity follow traditional programming languages and are summarized below:

Operator	Description	Associativity

`( )`
`[ ]`
`.`
	Parentheses (grouping)
	Brackets (array subscript and tupling)
	Member selection via object name
		left

`+ -`
`~`
`type:ex`
	Unary plus/minus
	Unary logical negation
	Type selection
		right

`* / %`	Binary Multiplication/division/modulus	left
`+ -`	Binary Addition/subtraction	left
`< <=` `> >=`	Relational less than/less than or equal to Relational greater than/greater than or equal to	left
`= ~=`	Relational is equal to/is not equal to	left
`**`	Exponentiation	right
`/\`	Logical and	left
`\/`	Logical or	left
`=> <=>`	Logical implication and equivalence	left
`\A \E`	Universal and Existential quantifiers	left

Invariant Scoping

Invariant scopes are not limited to the automaton they reference. For example, the Tempo model below

```
automaton A
 signature output out
  states x: Bool
 transitions output out

automaton B
 signature output out
  states x: Bool
 transitions output out

invariant of A:
 A.x;
 B.x
```

used to report an error on the last line because B was unknown. The current implementation finds B's definition in the "parent" (global) scope and type checks the model correctly.

Choose statement

Choose statements require that the chosen variable's type exactly equals the type of the left hand value. For instance, in the Tempo model

```
automaton A
  signature output act
  states chosen: Int
  transitions
    output act
      eff
        chosen := choose x where 1 <= x /\ x <= 30;      <-- x must be an Int
        chosen := choose x:Real where 1 <= x /\ x <= 30; <-- x cannot be a Real
```

The new checker rejects the last choose statement because the modeler forced x to be of type Real which is a super type of chosen's type.

Differential equations in trajectories

1. Overloading: The new checker does not allow the identifier 'd' to be used. It is reserved for the derivative function. The old checker only prohibited overloading of 'd' as a Real -> Real function

2. Type Spec: The new checker extends the type spec of 'd' from Real -> Real to AugmentedReal -> AugmentedReal. A number of examples from the Simulator used this construction.

IF-THEN-ELSE expression

This syntactic form is now obsoleted in favor of the more traditional lower case variant if-then-else

Implicit Type Declarations

In the new checker no types are generated implicitly. All types must appear in a type definition construction beginning with the key word "types", ether in a vocabulary or in a global type declaration. For instance, the declaration

```
Automaton A(type M) ....

Automaton B
   components   A(Square)
   ...
```

will no longer introduce a type 'Square' implicitly. Instead, the modeler is requested to explicitly declare which names denote types. To this end, the type definition construction is now authorized at the top level. The example above could be rewritten as

```
types Square;
Automaton A(type M) ....

Automaton B
```

```
components A(Square)
...
```

Type Selectors

Type selection used to be important to manually assist the front-end in resolving type ambiguities when several objects of the same name but with different types coexist. The old front-end used to attempt a type selection based on the typing context and the known objects. The new front-end implement stricter scoping rules (resulting in fewer instances of ambiguous constructions) and is equipped with a more flexible type inference mechanism capable to resolve type ambiguities more often. As a result, manual type selection is becoming less and less useful and we strongly encourage modelers to refrain from using it.

Variable overloading

Variables can no longer be overloaded by type. In each scope an Identifier can only refer to one object.

Null vocabulary

The __.val method of the Null vocabulary was changed to a unary function val(__). For example an expression

```
x.val = nil
```

now becomes

```
val(x)= nil
```

Top level declarations

1. As stated before, types declaration can now appear at the top-level
2. Vocabulary import statements can also appear at the top-level (e.g., to import types needed to instantiate automata)

Floating-point literals

It is now possible to enter floating-point literals directly (e.g. 2.5467)

Scoping Rules

The scoping mechanism of the new checker is much more extensive. Previously, variables existed in the same scope and could be overloaded by using different types, and then selected using a type selector. Now variables cannot be overloaded in the same scope, but they may be redefined or shadowed in a nested scope, with the same type or a different type. This forces each identifier to be unique in a given scope. Note that several constructions automatically create scopes, for instance

- Vocabularies: Scope for vocabulary body

- Automaton definition: Scope for the automaton formal and a nested scope for the automaton body
- Action: Scope for action formal and a nested scope for the action body
- Quantifiers: Scope for the nested expression

Plugins Notes (v0.1.4)

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

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.

Composition

Composite automata are not supported.

Schedules

Schedule blocks are ignored.

2. UPPAAL

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, “const”, “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 (\A and \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,

```
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.

Signatures

Signature overloading is not allowed

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

```
invariant mode = ...
stop when time = ...
```

where "mode" variable is fixed to be one of the enum value of the Location type. 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 simulator imposes a certain number of restrictions on the core Tempo language. These restrictions are listed below and will be progressively removed as new releases come out.

Null Vocabulary

The simulator does not support the Null vocabulary. Any attempt to import this builtin vocabulary will result in a semantic error. This is a temporary restriction that will be lifted shortly.

State variables declarations

The declaration of a state variable must always include an initialization statement. For instance the fragment

```
automaton A
state
x : Int
```

will trigger an error whereas

```
automaton A
state
x : Int := 10;
```

is acceptable.

**Composition*

Automata composition is allowed with some restrictions.

1. The composed automaton and its components are restricted to specific types of parameters (see Automata Formal Parameters).
2. Components cannot be initialized as arrays. For example,

   ```
   components
       comp4[x:Int]:A where x < 100 /\ x > 10;}
   ```

   is not supported.
3. All restrictions that apply to single automaton specifications, also apply to composite specifications.

det statements

det statements are not implemented in the IR.

Simulation

Simulation is not implemented in the IR.

Tasks

Tasks are not implemented in the IR.

Dereferencing

This is only a temporary restriction. The simulator only allows a single dereferencing level, as demonstrated in this example. However, even with this restriction one can still access deeper levels with use of the local variables.

Expressions and quantifiers

Quantifier expression that the simulator can resolve have to based on enumerable types, examples of things that are not enumerable are:

```
AugmentedReal, Char, DiscreteReal, Int, Real, Seq, MSet, Null, Set, String
```

This means that the we only support quantifiers on:

```
Bool, Nat, and user Enum vocabs
```

Set Notation

The simulator accommodates the new set notation allowing sets of multiple values. Specifically,

```
x : Set[Nat] := {0,2,4,6,8}
```

Is now a valid set definition. However, the simulator imposes a restriction that an explicit set is at most ten items. If you would like to construct a explicit set with more than ten elements the union operation can be used to join multiple explicit sets. For example,

```
x := {1,2,3,4,5,6,7,8,9,10} \U {11,12,13}
```

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. Nat, Int, Real, DiscrteReal, AugmentedReal) and may be of the following form:

:= choose x where _____ (/\ or \/) _____

:= choose x where _____

where the _____ may be a simple relational operator (-=, =, >, >=, <, <=) with parameter x and a literal.

Each choose statement is associated with a pseudo-random number generator, where with each invocation of the choose statement (perhaps as a result of multiple execution of transition containing the statement) a number that is next in the sequence will be assigned to x. If a user is interested in obtaining a random value from the domain of variable that appears on the left-hand-side of the choose statement, then the following statement will do the trick:

:= choose x

To specify upper and lower bounds on the domain from which the random number is chosen, use the following:

:= choose x ____ /\ _____

where _____ are relational operators that specify upper and lower bound on the domain from which the number is chosen. We cannot guarantee that a number is returned with equal probability if the following is used:

:= choose x ____ \/ _____

Note that these are legal on all data types:

:= choose

Of course these are equivalent to simply not initializing the state variable at all. Choose statements may not appear in other parts of the specification.

global types

Global types have to be encapsulated by

```
vocabulary
    ...
end
```

Otherwise the simulator will not recognize them.

let definitions

Unfortunately, let definitions are not allowed. The simulator cannot find implementations for these functions and hence does not know how to simulate them.

Schedules

Simulator must have a schedule block in order to simulate any specification. If no schedule block is given then the only output that will be given to the user is that of the internal representation of the specification that is passed in to the simulator. However, the simulator will demand a schedule block if two conditions are true:

1. specification has trajectories
2. specification has parametrized actions

For-statements

Simulator supports only these for-statements that have as a predicate a condition on a set, for example:

```
for j:Nat in s do
    ...
od;
```

where s is a Set[Nat]. Any other predicate is not supported.

Evolve statement

The simulator currently restricts the evolve statements to the following format:

```
d(x) = lit
```

where lit may be an integer or a decimal value. All other evolve statements are not supported, some examples of these are:

```
d(x) > 5;
d(x) = x;  --- where x is a variable
```

Smart fire

Currently the simulator requires user to provide a schedule which includes fire statements that are followed by an action kind and name, with parameters if any are needed. Basically, it is not able to choose an enable action from the pool of all enabled action in a given state.

Transition terms

The parameters in the transition must be variables. Moreover, if a transition has more than one parameter, then the variables must have unique names. Note that the simulator will not complain when it is provided with a constant value, such as:

```
output out(10)
output out(true)
```

However, there are two issues, (1) providing a constant literal as a parameter does not make the transition unique and the first one will be called (as listed in the specification). (2) This is an example of a sloppy programming, since this parameter cannot be assigned to any variable inside the transition.

User defined operators in vocabulary

As it is the case with the let statements, these are not supported.

Automata formal parameters

Automata formal parameters specified in the formals file can only be of the following types:

1. Nat
2. Int
3. Real
4. DiscreteReal
5. AugmentedReal
6. Bool

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 the existing TempoMacro file, the "-makeMacro" command line option will force the existing macro file to be overwritten.

Enjoy!