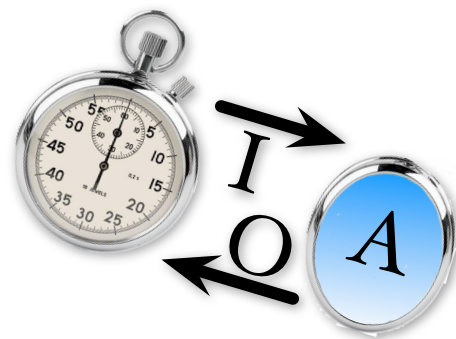


TEMPO RELEASE NOTES

vo.i.8 (BETA)

5/4/2007

Recent Fixes (vo.i.8)	3
*vo.i.8 - 5/4/2007	3
vo.i.7 - 4/27/2007	3
vo.i.6 - 3/9/2007	3
vo.i.5 - 1/31/2007	3
vo.i.4 - 12/14/2006	4
Known Issues (vo.i.8)	4
General	4
*Tempo Simulator	4
Graphical User Interface	4
Language changes (vo.i.8)	4
*Simulation Relations	4
*Simulate Block	5
*Type Cast	5
*Transition Locals	5
*Transition Case Labels	6
*Terminators and Separators	6
*Empty Automata	6
*Let Functions	7
*Type Shorthands	7
*Component Arrays	8



*Set Constructor	8	
*Simulate Block	8	
For Statements	9	
Through Notation	10	
String Literals	10	
Vocabulary Imports & Type Declaration		10
Imported Vocabulary Instantiation	11	
Built-in Vocabulary Changes	11	
Plugins Notes (vo.I.8)	12	
1. PVS	12	
2. UPPAAL	14	
3. *Simulator	15	
4. LaTeX Translator	19	

***vo.I.8 - 5/4/2007**

- Subsections of this document which have been updated in vo.I.8 or vo.I.7 have been marked with an asterisk (*).
- More changes were made to the Tempo syntax, see the Language Changes section of this document for details.
- The semantics of the set constructor { __ where __ } have been updated to support any type of element.
- The boolean operators in the simulator no longer use eager evaluation.
- The equality operators of the built-in types have been fixed to represent structural equality.
- Automaton where clauses are now checked by the simulator.
- The Sequence operator __[__] has been updated to be τ based, previously it was \circ based.
- The simulator now supports For loops in simulate blocks.

vo.I.7 - 4/27/2007

- Many changes were made to the Tempo syntax to make the language more uniform, see the Language Changes section of this document for details.
- A new simulator plugin is available with this release. See the Simulator Plugin Notes in this document for details.
- The new simulator plugin supports simulation of user defined vocabularies if a native java implementation is provided. See the Simulator manual supplement included in the docs folder for details.
- The PVS translator has been extended to support composite automata.
- The Tempo UI has been updated to support compiling Tempo models that span multiple files.

vo.I.6 - 3/9/2007

- There were several changes to Vocabularies, see the Language Changes section of this document for details.
- The PVS translator's integration with PVS 3.2 has been enhanced, see the PVS manual supplement included in the docs folder for details.

vo.I.5 - 1/31/2007

- The PVS translator now has an integration with PVS 3.2

vo.1.4 - 12/14/2006

- 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 been extended to support multiple elements. Please see the language changes section for more details.
- The simulator now supports follow statements in composite automata schedules.

Known Issues (vo.1.8)

General

- The automatic invariant validation mode of the new PVS integration is not yet implemented.
- Only the command line tool is supported by Windows Vista. This will be the case until the Eclipse Rich Client Platform has been ported to Windows Vista.

*Tempo Simulator

- The quantifiers \forall and \exists do not work correctly in Trajectory stop when clauses

Graphical User Interface

- Tab characters are not supported when using the Simulator in debug mode. If a file contains tabulations, the debugger may highlight an incorrect section of the source code.
- The windows distribution of the Tempo User Interface is designed and tested for Windows XP and is **not** supported by Windows Vista.
- (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 (vo.1.8)

*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’.

***Simulate Block**

The simulate block has been updated to support instantiation of simulations in the run command as follows,

```

forward simulation F(a1,a2:Int) where a1 ~= 0 /\ a2 ~= 0
  ...
simulate do
  run F(-5,5)
od

```

Note, although the Tempo language supports this construction it is not yet supported in the simulator.

***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 more 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 which 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;
  ...

```

*Transition Case Labels

Transition Case labels have been removed from the language. An example of the old syntax,

```
transitions
  output out case X
...
fire output out case X
```

*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 Clauses
- Statement Blocks

Comma separated

- Type Declarations
- Vocabulary Operators
- Vocabulary Imports
- arguments and formals of functions, automata, and actions

*Empty Automata

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

```
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

```
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

```
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 the integers to the integer and 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-know factorial function can be written as,

```
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,

```
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

```
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,

```
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

```
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 C_1 as an array of automatons $AutoA$ and there is one entry for each natural number j less than 10. The second line defines C_2 as an array of $AutoA$ with one instance for each value j that belongs to type $Color$. The third line states that C_3 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,

```
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}
```

*Simulate Block

Previous versions of the simulator relied on a “formal files” to specify the argument that should be used to instantiate an automaton before its simulation. This mechanism has been replaced by a language construction to specify and possibly script the instantiation of the automaton to simulate.

The simulate block is a list of statements that can use a special “run” command to instantiate an automaton and execute its schedule. Consider the following model

```
automaton A
  signature output out
  states x:Bool := true;
  transitions output out
  schedule do
    print x;
  od

automaton B(n:Int)
  signature output out
  states x:Int := n;
  transitions output out
  schedule do
    print x;
  od

automaton C(n:Int)
  components
    c1 : A;
    c2 : B(5);
    c3 : B(n);
    c4 : B(2*n);
  schedule do
    print c1.x;
    print c2.x;
    print c3.x;
    print c4.x;
  od
```

It specifies three automata (A,B,C) and defines C in terms of A and B. Tempo supports the specification of a simulation block to script several test. Note that, when the model contains a single ground automaton, Tempo synthesizes a simulate block automatically.

```
simulate do
  run A;
  run B(5);
  run C(10);
  for y in {1,2,3,4,5}:Set[Nat] do
    run B(y);
  od;
  print "done";
od
```

This example illustrates how to simulate A, then B (instantiated with the argument 5), then C(10) then simulate B for all the values of the argument between 1 and 5 before reporting that the simulation is completed.

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,

```
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

types Color enumeration [red, white, blue]
for y in Color 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,

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

yields the following set,

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

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

```
for y in -3..3 do ... od
```

String Literals

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

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

the exact lexical specification is,

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

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

Vocabulary Imports & Type Declaration

Vocabularies have been enhanced to support multiple imports and type declaration blocks. Previously vocabularies were limited to one import clause directly before type declarations. For example,

```
vocabulary Foo
  imports Bar(type Int, type Real)
  types MyType
  ...
```

However, some specifications were importing a vocabulary using a type defined after the import statement. For example,

```
vocabulary Foo
  imports Bar(type MyType, type MyType)
  types MyType
  ...
```

To support both needs, Tempo now allows multiple imports and type definitions. For example,

```
vocabulary Timestamp
  types myType
  imports Messages
  types TM : Tuple [message: M, timestamp: myType]
  imports Char
  ...
```

Imported Vocabulary Instantiation

Parametric vocabularies can no longer be imported without instantiation. Previously, the following specification fragment was acceptable,

```
vocabulary Foo defines Foo[T]
...
vocabulary Bar(T1, T2 : type)
...
automaton A
imports Foo, Bar
```

Parametric vocabularies must now be imported with fully qualified instantiation. The example above should be rewritten as follows,

```
automaton A
imports Foo(Int), Bar(Int, Int)
```

If you prefer or need to retain abstract types, the example should be corrected as follows,

```
automaton A(M : type)
imports Foo(M), Bar(M, M)
```

Built-in Vocabulary Changes

Some of the built-in vocabularies were re-named so that they would follow the implicit loading convention that the vocabulary name is the same as the type it defines. The built-ins which were re-named in old name - new name format,

- Boolean - Bool
- Character - Char
- Integer - Int
- Multiset - Mset
- Natural - Nat
- Sequence - Seq

This change will be totally transparent with one exception. If a built-in vocabulary is explicitly imported the name of the imported vocabulary will need to be changed to the new vocabulary name (aka the type name).

Additionally the the NumericOps built-in vocabulary has been removed. Similar to the built-in name change. This change will be totally transparent with the exception of explicit importing of NumericOps.

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.

I. PVS

PVS Integration

The PVS Translator plugin now supports an integration with the PVS 3.2 application. After translation is complete the plugin can start PVS and begin proving some 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.

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 “incr” 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 “incr” 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 “incr: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

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 (\forall and \exists)
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 $d(t)=\tau$ (time evolves at constant rate τ). 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 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 see the release archives at www.VeroModo.com/tempo. The restrictions that are listed below will be progressively removed with new releases.

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

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.

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

:= choose x where _____ (\wedge or \vee) _____

:= choose x where _____

where the _____ may be a simple relational operator ($\neq, =, >, \geq, <, \leq$) 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 _____ \wedge _____`

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 _____ \vee _____`

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.

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 its 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:

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

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

Evolve statement

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

```
d(x) = <literal value>;
```

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

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

Smart fire

Currently the simulator requires 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 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 manuel supplement included in the docs folder.

Type Shorthands (Tuple, Enumeration, Union)

The simulator supports all type shorthands if they are defined in a vocabulary or global type declaration. Only the Tuple and Enumeration type shorthands are support for anonymously defined type short hands. For example,

```

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

  automaton A
    states
      colors : Set[Enumeration[red, green, blue]]; %ok for simulator
      pair   : Tuple [p: Int, s: Int];           %ok for simulator
      numList : Seq[Union[i: Int, r: Real]];      %not supported
    end
  end

```

Ignored Constructions

There are several constructions which the simulator will ignore. These are

- Simulation Blocks (aka forward simulation and backward simulation)
- Task Blocks
- Smart Fire statements
- Composite Automaton Hidden 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 the existing TempoMacro file, the “-makeMacro” command line option will force the existing macro file to be overwritten.

Enjoy!