The TCOB Programmer's Manual 1.1

Aswathy M.S
Amrita School of Computing, Amritapuri
Amrita Vishwa Vidyapeetham

Jinesh M.K
Amrita School of Computing, Amritapuri
Amrita Vishwa Vidyapeetham

Bharat Jayaramam
Amrita Institute of Advanced Research, Elburn, Illinois 60119

February 14, 2025

1 Introduction

TCOB is a declarative modeling language based on the concept of temporal constrained objects. Constrained objects [1] were proposed as a modeling framework for complex systems that consist of an assembly of interconnected, interdependent components. This paradigm uses object-oriented concepts such as classes, inheritance, and aggregation for structural modeling and declarative constraints for behavioral modeling. Temporal constrained objects [2] add a temporal dimension to constrained objects and facilitates the declarative specification of time-dependent behavior in complex systems. The key new feature is that of a *series variable*, which records the temporal sequence of values bound to a variable. Temporal constraints may be placed over the consecutive values of a series variable. TCOB allows temporal constraints through the introduc- tion of metric temporal operators, which are metric variants of the classic F and G operators of linear temporal logic (LTL). The execution model for temporal constrained objects involves a time-based simulation along with constraint solving at each time-step. This document describes the syntax of TCOB programs and also the steps for compilation for visualization of program execution.

2 TCOB Syntax

A TCOB program defines a collection of classes each of which incorporates a set of attributes, constraints, predicates, and constructors. We use the .tcob extension for source files. A program starts with optional meta information enclosed in $\{\}$ in order to define the simulation start and end times, and the debug option.

program ::= *{meta_info}* *class_definition*$^{+}$\$

meta_info ::= *{*[simulation start = *value*1*,*] [simulation end = *value*2*,*] [debug = *value*3]*}*

The specification of meta information is optional as default values are provided as shown in the table below.

Table 1: Meta-information parameters

Parameter	**Description**	**Value**	**Default**
simulation_start	define simulation start time	>=1	1
simulation_end	define simulation end time	>simulation_start	10
debug	Enable or disable the debug option	yes/no	no

A TCOB program must end with a \$ sign and text appearing after it is ignored by the compiler.

Class. Every object is an instance of some class whose outline is as follows.

class_definition ::= [abstract] class *class id* [extends *class id*]*{ body }*

body ::= [attributes *attributes*]
[constraints *constraints*]
[predicates *predicates*]
[constructors *constructors*]

Attributes. An attribute is a typed identifier, and the language supports both primitive and user-defined types. In the case of series variable, the series keyword is used before the type declaration.

attributes ::= [*decl* ;]$^{+}$
decl ::= [series] *type* *id_list*
type ::= *primitive* | *class id* | *type*[ ]
primitive ::= real | int | bool | char | string
id_list ::= *attribute_id* [*,* *attribute_id*]$^{+}$

Constraints. Constraints are relations over the attributes of classes. TCOB supports simple, conditional, quantified and creational constraints over typed attributes.

constraints ::= [*constraint* ;]$^{+}$
constraint ::= *creational* | *quantified* | *simple*

A simple constraint can either be a constraint atom or a conditional constraint.

$$simple \ ::= \ conditional \mid constraint\_atom$$

Constraint Atom: A constraint atom is a relational expression of the form *term relop term*. A constraint atom may also make use of a predefined constraint predicate whose properties are known by the underlying system.

$$\begin{aligned} constraint\_atom \ &::= \ term \ \ relop \ \ term \mid cpred\_id(terms) \\ relop \ &::= \ = \mid \ != \mid > \mid < \mid >= \mid <= \end{aligned}$$

Term: A term may be an arithmetic or boolean expression involving attributes, constants from the primitive types, and built-in functions. Aggregate operations over the elements of an array or elements of an explicitly specified set can also be specified, as follows.

$$\begin{aligned} term \ &::= \ const \mid var \mid attribute \mid (term) \mid fun\_id(terms) \mid \\ &\quad\ aggr \ var \ \text{in} \ enum \ : \ term \\ aggr \ &::= \ \text{sum} \mid \text{prod} \mid \text{min} \mid \text{max} \end{aligned}$$

Literals: A literal is an atom or negation of atom.

$$\begin{aligned} literals \ &::= \ literal \ [, \ literal]^{+} \\ literal \ &::= \ [\ \text{not}\ ] \ \ atom \end{aligned}$$

Conditional Constraint: A conditional constraint is a constraint atom that is predicated upon a conjunction of literals each of which can be a constraint atom or ordinary atom or metric temporal constraints, or their negation. The constraints specified with metric temporal operators are called metric temporal constraints.

$$\begin{aligned} conditional \ &::= \ literals \ \ {-}{-}{>} \ \ condition\_body \\ condition\_body \ &::= \ MTOliterals \ [\& \ MTOliterals]^{+} \\ MTOliterals \ &::= \ literal \mid MTOconstraint \\ MTOconstraint \ &::= \ \text{F} \ constraint\_atom \mid \text{F} < time > \ constraint\_atom \mid \\ &\quad\ \text{F} < time, time > \ constraint\_atom \mid \text{G} \ constraint\_atom \mid \\ &\quad\ \text{G} < time > \ constraint\_atom \mid \\ &\quad\ \text{G} < time, time > \ constraint\_atom \end{aligned}$$

A conditional constraint of the form $P \ {-}{-}{>} \ Q$ is only evaluated when P is ground, i.e., there are no unbound variables in P. If P is not ground, the evaluation of the conditional constraint is postponed until P becomes ground.

Quantified Constraint: The quantified constraint states a relation over an enumeration, i.e., indices of an array or the elements of an explicitly specified set.

$$\begin{aligned} quantified \ &::= \ \text{forall} \ \ var \ \ \text{in} \ \ enum \ : \ constraint \mid \\ &\quad\ \text{exists} \ \ var \ \ \text{in} \ \ enum \ : \ constraint \end{aligned}$$

The quantification can be either universal or existential. The universal con- straint must be satisfied by every member in the enumeration while the exis- tential constraint must be satisfied by some member of the enumeration.

Creational Constraint: The creational constraint creates and object of a user-defined class by using the new keyword.

$$\mathit{creational} \ ::= \ \mathit{attribute} \ = \mathrm{new} \ \mathit{class\_id}(\mathit{terms})$$

A creational constraint is satisfied by invoking the constructor of the class with terms as arguments.

Predicate. A user-defined predicate stands for an n-ary relation over terms and is defined using Prolog-like rules defined in the standard *Head :- Body* format.

Constructor. A class that is not abstract can have one or more constructors.

$$\begin{aligned} \mathit{constructor\_clauses} \ &::= \ \mathit{constructor\ clause}^{+} \\ \mathit{constructor\_clause} \ &::= \ \mathit{constructor\ id}(\mathit{formal\ pars}) \\ &\qquad \{\ \mathit{constraints}\ \} \end{aligned}$$

The type of a formal parameter of the constructor is not specified. Hence selection operation on these parameters is not allowed. However, if they are equated to a local attribute of the class, then the select operation can be performed on the local attribute, since its type is known. The constructor id must be the same as the name of the class. The body of a constructor contains a sequence of ; separated constraints.

3 Examples

AC Circuit: The voltage in AC circuits vary with time, typically alternating between positive and negative values according to some pattern. We define the voltage (V) and current (I) in the abstract class component as series variables to model this periodic change.

```
abstract class component {
  attributes
     series real V, I;
}
```

The source class generates the input voltage for the circuit, which varies sinusoidally with time. It extends the component class and makes use of a in-built sine function for input generation.

```
class source extends component {
  constraints
     sin(Time,V);
  constructors source()
    { }
}
```

The three types of circuit components (resistor, capacitor, and inductor) each inherit voltage and current from the component class. The constraints in each class define the laws of circuit elements: the resistor class incorporates Ohm's law for resistance; the inductor class incorporates Faraday's law of inductance, i.e., the voltage (V) across an inductor is equal to the inductance (L) multiplied by the rate of change of current (I); similarly, the capacitor class incorporates Ampere's law of capacitance, i.e., the current (I) through a capacitor is equal to the capacitance (C) multiplied by the rate of change of voltage (V).

$$V = I \times R$$

$$V = L \times dI/dt$$

$$I = C \times dV/dt$$

The TCOB simulation evaluates the constraints at each time-step. That is, the rate of change of current (voltage) can be approximated by the change in current (voltage) in one unit of time[1]. In other words, the differential equations are treated as difference equations.

```
class  resistor
       extends component{
 attributes
    real R;
 constraints
    V = I * R;
 constructor resistor(R1)
    { R = R1; }
}

class  inductor
        extends component {
 attributes
    real L;
 constraints
    V = L * (I – 'I);
 constructor inductor (L1)
    { L = L1; I<1> = 0.0; }
}

class  capacitor
        extends component {
 attributes
    real C;
 constraints
    I = C * (V -'V);
 constructor capacitor(C1)
    { C = C1; V<1> = 0.0;}
}
```

The series2 and parallel classes represent two different ways to combine circuit elements[2], and the constraints define the effective voltage and current for

[1]One can use a finer granularity for time by defining a series variable MyTime = Time * 0.1, for example. The voltage source would then be defined as sin(MyTime, V), and the constraints in class inductor and capacitor would use 0.1 as the denominator on the RHS of the equations.

[2]It is of course, possible to define non-series-parallel circuits. These would require the definition of a node class which incorporates Kirchoff's law for currents.

these two forms of circuits. Consider the parallel class: the forall constraint specifies that the voltage through every component of the parallel circuit is the same and equal to the voltage across the parallel circuit; and, the sum constraint sums up the current through each component in the parallel circuit and equates it to the current across the parallel circuit.

```
class series2 extends component
{
  attributes
    component [] SC;
  constraints
    forall C in SC: C.I = I;
    (sum C in SC: C.V) = V;
  constructor series2(A)
     { SC = A; }
}
```

```
class parallel extends component
{
  attributes
    component [] PC;
  constraints
    forall X in PC: (X.V = V);
    (sum X in PC: X.I) = I;
  constructor parallel(B)
     { PC = B; }
}
```

The samplecircuit class creates instances of the different circuit elements and connects them together to form a simple circuit, as shown in Figure 1.

```
class samplecircuit {
  attributes
    source AC;
    resistor R1,R2;
    inductor L;
    capacitor C;
    series2 S;
    parallel P;
    component[] Ser;
    component[] Par;
  constructor samplecircuit() {
    R1 = new resistor(10.0);
    R2 = new resistor(10.0);
    L = new inductor(1.0);
    C = new capacitor(0.1);
    Ser[1] = R1; Ser[2] = L;
    Ser[3] = C;
    S = new series2(Ser);
    AC = new source();
    Par[1] = S; Par[2] = AC;
    Par[3] = R2;
    P = new parallel(Par);
  }
}
```

The voltage source now generates a sine wave and is connected in parallel with resistor and a series arrangement of a resistor, inductor, and capacitor. The source file ac circuit.tcob is available in the TCOB/Examples/Basic folder. It starts with the following meta-information:

```
{simulation_start=2,  simulation_end=10}
```

Simple Traffic Light: This is a simple example of the use of metric temporal operators to model a traffic light, especially to specify the criteria for transitioning from one color to the next. The constraints state that the traffic light starts with a red light, stays at red for 120 time units, then changes to yellow for 20 time units, then changes to green for 180 time units, then changes to yellow for

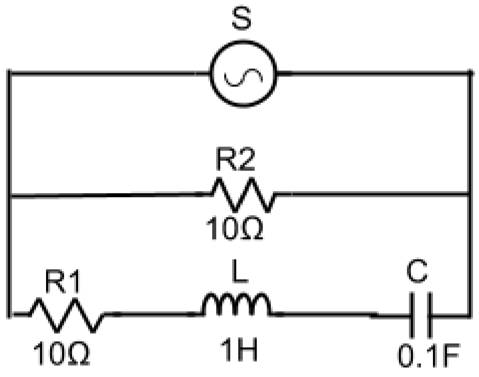


Figure 1: Simple Circuit

40 time units, and changes back to red. This behavior is repeated until the end of the simulation. The metric temporal operators in conjunction with the series variables enables a concise and clear specification of temporal constraints in a real-time setting.

```
class trafficlight {
      attributes
          enum Color;
          series Color C;
      constraints
          C = red & not (‘C = red) -->
                 G<0,120> C = red  &  F<120> C = yellow;
          C = green & not (‘C = green) -->
                 G<0,180> C = green  &  F<180> C = yellow;
          C = yellow &  ‘C = red -->
                 G<0,60>C = yellow  &  F<60> C = green;
          C = yellow & ‘C = green -->
                 G<0,40> C = yellow   &  F<40> C = red;
     constructor  trafficlight()   {   Color=[’red’, ’green’, ’yellow’];
          C<1> = ’yellow’; C<2> = ’red’; }
}
```

The complete program available in simple traffic.tcob (TCOB/Examples/Basic folder).

4 Compiling and Running TCOB programs

The compiler translates a TCOB program into a CLP(R) program in the SWI-Prolog environment. In order to work with the TCOB compiler, you should first download and install SWI-Prolog from
http://www.swi-prolog.org/download/stable.
The user-manual and documentation of SWI-prolog is available at
http://www.swi-prolog.org/pldoc/index.htm.
The compiler contains following Prolog files (available in TCOB/Compiler).

1. tcob2swi.pl - TCOB compiler source, which translates a TCOB program into a CLP program

2. helper_clpr.pl - A collection of TCOB built-in predicates.

The following section describes steps to compile the ac circuit.tcob program. Here ?- is the Prolog prompt for commands.

Step 1 Open SWI-Prolog environment and load the compiler.

```
$swipl
?- [tcob2swi].
```

Step 2 Compile TCOB using the tcob2swi/2 command. It takes the TCOB program and driver class constructor as the input and creates the corresponding CLP(R) program.

```
?- tcob2swi('ac_circuit.tcob', 'samplecircuit()').
```

It creates a file with a .pl extension and with same name as the TCOB program.

Step 3 Load the compiled code using standard Prolog load command.

```
?- [ac_circuit].
```

The compiler adds a main class to regulate the simulation time based on user-supplied parameters and it invokes the driver class constructor. The generated program has two arguments, the first denotes the list of attributes in the class and second denotes the list of arguments to the constructor. If you are not taking any input values, it can represent by character.

```
?- main(_,_).
```

The result will be 'true' for a successful execution (all the constraints are satisfied), otherwise the result will be 'false'.

To know the values of variables, we can use the following methods.

Using write predicates TCOB supports the write/1 and writeln predicates of SWI-Prolog in order to print the value of variable. The user may use

these predicates in constraints, predicates and constructors of a TCOB class.
Example:

```
class  resistor
        extends component{
 attributes
     real  R;
 constraints
     V = I * R;
     writeln('I);
 constructor resistor(R1)
     { R = R1; }
}
```

Using constructor parameters A constructor can accept any number of parameters which can be used in order to get the values of variables after the execution. Consider the following modification in the samplecircuit class in order to get the value of **I** from the resistor class.

```
class samplecircuit {
  attributes
    source  AC;
    ....
  constructor  samplecircuit(RI)  {
    R1 = new  resistor(10.0);
    ....
    RI  =  R1.I;
  }
}
```

The compilation steps are as follows.

```
 ?- tcob2swi('ac_circuit.tcob', 'samplecircuit(RI)').
 ?-  [ac_circuit].
```

The value of RI after execution can be obtained as follows:

```
?-  main(_,[R1]).
R1 =  [0.028415544588302557,
          -0.0026938858952047884,-0.0318564641884161,-0.030040285877712104
          0.0006954638679985727,   0.031661681297034526,   0.03408940847804451
          0.005549458734292934]
  true.
```

5 Run-time Analysis

Run-time analysis refers to methods and tools for monitoring the run-time behavior of a program with the goal of debugging and verifying its behavior. We provide constructs to extract a TCOB program's execution trace, which is subsequently input to a web-based GUI in order to visualize the trace in more readable form. In order to obtain the execution trace, the body of every class definition may include an optional 'monitor' clause which specifies a list of class attributes that are to be monitored during execution. This clause is to be used in conjunction with the specification debug = yes in the meta-information clause.

```
body ::= [ attributes  attributes ]
         [ constraints  constraints ]
         [ predicates  predicates ]
         [ constructors  constructors ]
         [ monitor  attribute list ]
```

Here, *attribute list* is a comma-separated list of attribute names, and they can be both series variables as well as non-series variables. At run-time, the underlying system creates an execution trace file log.txt, which contains, for every time-point of execution, the object reference, attribute name and value for every designated attribute in every monitor clause in the program. Below we show the monitor clause in the inductor class.

```
 class inductor extends component {
 attributes
    real L;
 constraints
    V = L * (I - 'I);
 constructors inductor (L1) {
    L = L1;
    I<1>  = 0.0;
 }
 monitor V,L;
}
```

In order to invoke the program, instead of two parameters, main now takes three parameters:

```
?- main(_,_,_).
```

The file log.txt is created in the same folder as the TCOB program. The following shows a sample log file for the AC circuit example.

```
Time = 2: Obj = L: Var = V: Val = NaV
Time = 2: Obj = L: Var = L: Val = 1.0
Time = 2: Obj = L: Var = V: Val = 0.0432998774678896
Time = 3: Obj = L: Var = V: Val = NaV
Time = 3: Obj = L: Var = L: Val = 1.0
Time = 3: Obj = L: Var = V: Val = -0.05513696742746766
Time = 4: Obj = L: Var = V: Val = NaV
Time = 4: Obj = L: Var = L: Val = 1.0
Time = 4: Obj = L: Var = V: Val = -0.03974707481902298
```

Visualization serves as the cornerstone of our system's debugging and verification capabilities through an interactive web-based interface. Our implementation begins with a streamlined file upload process, where users can drag-and-drop or select their files through an intuitive interface. These files are then processed through PlantUML to generate abstract syntax representations, which are subsequently transformed into an interactive graph structure. The visualization engine, built with React and GraphViz, renders these graphs using React Flow for a dynamic and explorable representation. This approach enables users to examine complex relationships and structures within their data through an interactive visualization that supports features like zooming, panning, and node manipulation. The system incorporates a modern UI design using shadcn/ui components and Lucide icons, ensuring both functionality and aesthetic appeal in the visualization process. To run our interface, steps to be followed:

1.We need to install nvm(node version manager) to manage nodes and npm version for our interface.

Link to install nvm: https://www.freecodecamp.org/news/node-version-manager-nvm-install-guide/

2.Cloning the repository

a) On GitHub, navigate to the main page of the repository.

b) Copy the URL for the repository.

https://github.com/jineshmk/TCOB.git

c) Open Git Bash.

d) Change the current working directory to the location where you want the cloned directory.

e) Type git clone, and then paste the URL you copied earlier.

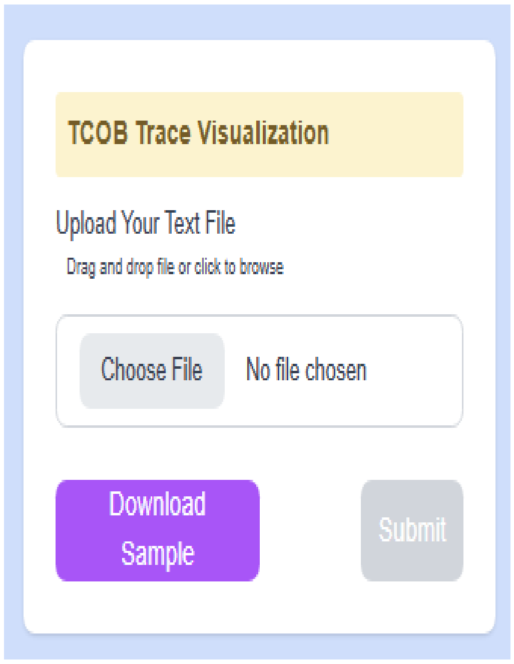
3. Now we start the web interface, navigate to TCOB/Runtime-New using Git cmd

and the following commands.

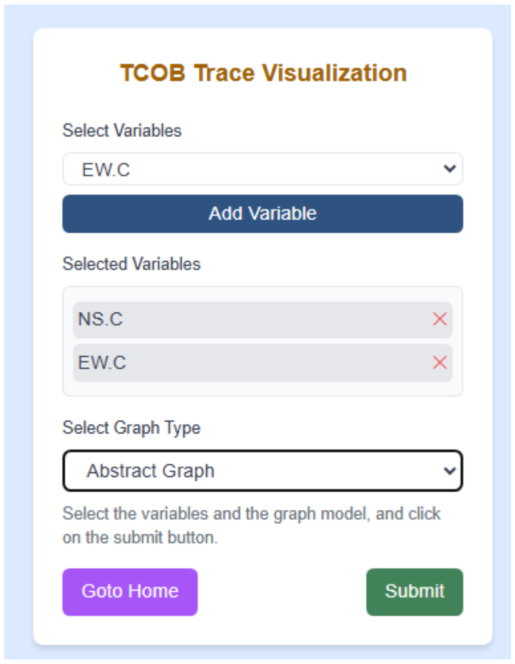
a) git pull

b) npm run dev

This command starts the server and the interface is available in the browser at URL
http://localhost:3000/.



b) Upload



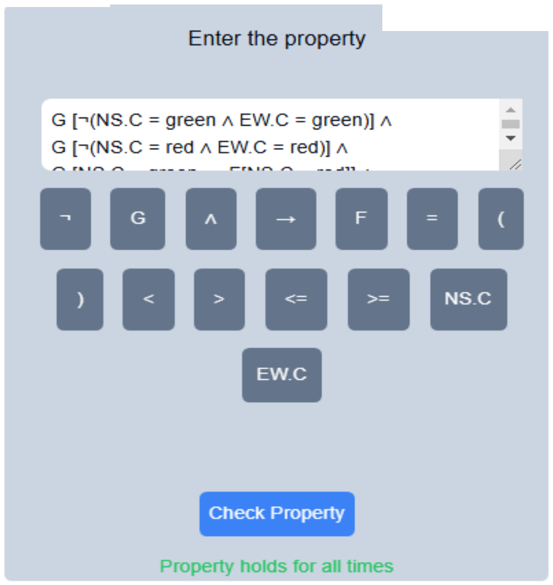
Select Variables: EW.C

Selected Variables: NS.C, EW.C

Select Graph Type: Abstract Graph

Select the variables and the graph model, and click on the submit button.

a) Selection of attributes



G [¬(NS.C = green ∧ EW.C = green)] ∧
G [¬(NS.C = red ∧ EW.C = red)] ∧

Property holds for all times

c) Property Verifier

Figure 2: Web Interfaces

Figure 2 shows the user interface, where the user can upload (Figure 2.a) an execution trace file generated by the monitored program and then select one or more attributes(Figure 2.b) for visualization. Currently, the framework supports three types of diagrams and a property verifier:

Timed State Diagram. A timed state diagram is a precise way to portray the

evolution of a system over time. This state diagram shows the linear progression of the system with respect to time.
Abstract State Diagram. The abstract state diagram shows the abstract view of the system in terms of the values of the selected variables. This diagram will be cyclic when there are repetitions in the values of selected variables.
Timed Line Diagram. The timed line diagram plots the values of the chosen variables over time. Unlike other diagrams, it supports only numerical attributes but is often useful in identifying incorrectness by direct inspection of the form the output diagram.
Property Verifier. In order to check the truth value of a propositional1 formula at some state of a linear or abstract state model, the state vector of values is first assigned to variables that stand for the respective object fields of the state vector. With this assignment of values to variables, the propositional formula is evaluated to either true or false. A temporal formula is checked with respect to a finite- state state model. We construct two types of runtime finite-state models: the linear state model and the abstract state model. For verification, we use the abstract state model, which is generally more compact for model checking.

6 Errors and Warnings

The TCOB compiler will point out a syntax error by naming the class in which it occurs, the attribute/constraint/constructor definition in which it occurs, and the index of the constraint and attribute declaration. If the error is in the i^{th} constraint, it means, it is in the i^{th} semi-colon separated constraint. The Prolog file is generated only when compilation is successful. Errors should be corrected and the TCOB compiler re-run. Warning messages related to singleton variables in the translated program will be reported by the Prolog compiler, but these can be safely ignored.

References

[1] B. Jayaraman and P. Tambay. Modeling Engineering Structures with Constrained Objects. In S. Krishnamurthi and C. Ramakrishnan, editors, *Symp. on Practical Aspects of Declarative Languages*, pages 28–46. Springer-Verlag, 2002.

[2] J. M. Kannimoola, B. Jayaraman, P. Tambay, and K. Achuthan. Tempo- ral Constrained Objects: Application and Implementation. In *Computer Languages, Systems & Structures*, volume 49, pages 82–100. Elsevier, 2017.