FlowTeX: A Domain Specific Language for UML Diagrams

Ashwin Rao
ORCID: 0000-0002-1545-2611
International Institute of Information Technology, Hyderabad
India
ashwin.rao@students.iiit.ac.in

Venkatesh Chopella
ORCID: 0000-0000-0000-0000
International Institute of Information Technology, Hyderabad
India
venkatesh.choppella@iiit.ac.in

ABSTRACT

UML class and state diagrams are an easy way to visualise relationships between entities in modular systems. They are used extensively in modelling software workflows in both professional and educational settings. However, existing solutions to create UML diagrams are either no-code or inefficient to use, or both. There exists no simple method to write readable code - similar to general purpose programming languages like Java or Python - specifically tailored to generate UML class and state diagrams. We propose a Domain Specific Language (DSL), **FlowTeX**, for this purpose. FlowTeX's syntax is heavily inspired by the functional programming language *Racket*. It defines classes, attributes, methods, parameters, and relationships, as well as states, substates and transitions as expressible values, introducing an efficient way to represent top-level design of modular systems in code. FlowTeX transpiles this concrete syntax into LaTeX code of the *TikZ-UML* package, which can be directly copy-pasted into any LaTeX document and visualized out-of-the-box. This makes our DSL an ideal tool to insert UML diagrams into any research paper or report without going through the hassle of exporting as an image. The generated code produces sharp graphics superior to images exported from no-code solutions.

1 INTRODUCTION

Software engineers spend a significant amount of time creating UML diagrams to visually demonstrate relationships between classes and modules, especially in **Object Oriented Designs**. Usually, they use online GUI tools for this purpose, such as *LucidChart*. This method has two issues. First, many of these tools follow a freemium model (*Creately*, for example) which hide most non-basic functionality behind a paywall. Second, and more importantly, these tools use no-code, drag-and-drop based workflows to generate UML and other diagrams. This method is suited to those with limited programming and technical knowledge. For a programmer, however, this approach is tedious and time-consuming. No-code tools are also difficult to customise. Being able to write code to create such diagrams would greatly improve the efficiency of the workflow for creating these diagrams. Our target audience is software researchers or developers making **software documentation** with basic coding knowledge - those who can leverage a simple API to create visual diagrams faster than existing GUI methods.

Our main aim is to reduce the amount of time spent on creating UML class and state diagrams in documentation - grunt work that can better be replaced with more important tasks. FlowTeX's syntax remains close to that of standard programming languages, meaning that it is extremely quick to pick up - unlike competing solutions described in Sec. 2. FlowTeX leverages existing solutions to generate high-quality LaTeX graphics which can be dropped into any existing project.

2 RELATED WORK

There exists a library to create diagrams (*TikZ-UML* [1]) in LaTeX. However, the syntax of this library is confusing and clunky. In particular, the library sacrifices good syntax for generalisability - it supports a wide array of diagrams from UML to graphs to flowcharts. It does not have a framework specifically for UML diagrams. This makes it inefficient to use and provides significant scope for improvement if we narrow the domain to UML. FlowTex focuses specifically on UML class and state diagrams, introducing a domain-specific syntax adapted to expressing classes, attributes, relationships, states and transitions in an efficient manner. FlowTeX abstracts out concepts such as associations and inheritance relationships between classes, and transitions between states and substates, in a natural manner.

Other solutions include making UML diagrams using online no-code tools (like LucidChart [2], Creately [3]), exporting the diagram as an image, and adding that image to the latex document. Here, the user has to worry about exported image quality, image size, etc. Since FlowTeX generates copy-pastable graphics-producing code, these issues are irrelevant.

There is also an open-source service *PlantUML* [4] which uses its own DSL to visualise UML diagrams. Although this service solves the issue of having to use no-code tools, it does not transpile to code but directly produces an SVG image, leading to the issues described above. By compiling to code rather than generating an image file, FlowTex leaves all graphics rendering to the LaTeX compiler.

3 SOLUTION DESIGN

FlowTeX is to UML diagrams as SQL is to tables. A FlowTeX user will descibe a UML class or state diagram in custom FlowTeX syntax. FlowTeX compiles and evaluates the input program into LaTeX code. This code can be put into any LaTeX document or research paper and compiled into the corresponding UML diagram.

3.1 Modelling UML Class Diagrams

FlowTeX provides the following functionalities to create UML class diagrams:

3.1.1 *Creating classes.*

- Specifying the class name.
- Specifying class attributes: name, datatype, access specifier (public, private, protected).
- Specifying class member functions: name, return type, input parameters (each with its own name and datatype).

3.1.2 Modelling relationships. Relationships are represented in the UML class diagram as different types of arrows. We model each relationship type as a separate expressible value.

- Inheritance between parent and child class.
- Association between classes with different cardinality and semantics: exactly zero, exactly one, zero or one, zero or more, one or more, and so on.

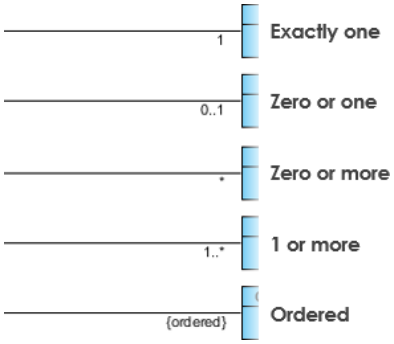

Figure 1: Types of associations between classes (from *Visual Paradigm* [5]).

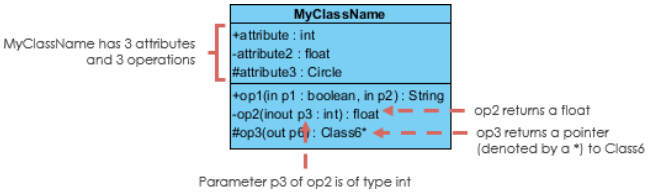

Figure 2: A sample UML class component in a diagram and its meaning (from *Visual Paradigm* [5]).

3.2 Modelling UML State Diagrams

FlowTeX provides the following functionalities to create UML class diagrams:

3.2.1 Creating states.

- Specifying the state name and label.
- Specifying special states (initial, final).
- Recursively specifying substates within a state.

3.2.2 Modelling state transitions. Transitions are represented in the UML state diagram as labelled arrows connecting states. We model a transition as an expressible value consisting of a "to" state, "from" state, and label.

3.3 Positioning System

FlowTeX uses a general absolute positioning system to specify the location of any class, state or substate in X and Y coordinates. The positioning system is kept separate from the logic of modelling classes, states and transitions, as seen later in FlowTeX's concrete syntax (Sec. 4.1).

4 CONCRETE AND ABSTRACT SYNTAX

In this section, we describe FlowTeX's concrete and abstract syntax and the design choices behind them.

4.1 Concrete Syntax

The concrete syntax of FlowTeX is shown in Fig. 3. The syntax is inspired by functional untyped racket.

```
program := flowtex
flowtex := (class | associate | inherit | state | basicstate | transition | position ...)

class := ('class classname attributelist methodlist)

attributelist := ('attributelist (attribute ...))
attribute := (access-specifier name datatype)

methodlist := ('methodlist (method ...))
method := (access-specifier returntype name ('params (param ...)))
param := (datatype name)

associate := ('associate (classname1 classname2) (cardinality1 cardinality2))
inherit := ('inherit classname 'from classname)

state := ('state statename label (state | basicstate | transition ...))
basicstate := ('state statename statetype)
transition := ('transition statename 'to statename label)

position := ('position name x y)

access-specifier := public | private | protected
type := initial | final
classname, name, statename, statetype := <string>
label := "<string>"
datatype, returntype := <string>
x, y := <number>
```

Figure 3: Concrete syntax and grammar rules of FlowTeX.

4.1.1 UML Class Diagrams. As seen in sample FlowTeX programs, the definitions of attributes, class methods and function parameters follow semantics similar to C (access specifier followed by datatype followed by variable name). The program is modelled as a sequential list of classes and relationships, defined in any order.

- A class is identified by the *class* keyword and has a name, list of attributes (attributelist) and a list of methods (methodlist).
- The list of class attributes is identified by the *attributelist* keyword and is defined as a list of the *attribute* type, which is a 3-tuple consisting of an access specifier, name and datatype.
- The list of class methods is identified by the *methodlist* keyword. It is defined as a list of the *method* type. A method is a fixed-length tuple consisting of an access specifier, return type, method name, and a list of parameters identified by the *params* keyword. Each parameter is 2-tuple consisting of the parameter's datatype and name.
- The *associate* expression models all types of associations between classes, as described in Sec. 3.1.2. As an association is between two classes, this expression takes the two associated classes as input, along with the two cardinalities of the association.
- Similar to association, the *inherit* expression models inheritance between a parent class and child class. This type of expression is identified by the keyword *inherit*.

4.1.2 UML State Diagrams. A FlowTeX program for a UML state diagram is modelled as a sequential list of states and transitions, defined in any order.

- A state is identified by the *state* keyword and has a name, label, and a list of substates and transitions between them.
- Special states are identified by the *basicstate* keyword. Each has a name and type (specifying the type of state, such as initial state and final state).
- The *transition* expression models state transitions. As a transition is between two states, this expression takes the "to" state, "from" state, and a label.

4.1.3 Positioning. Positioning of a class or state is done using the *position* keyword. The expression for positioning requires the name of the item to be positioned and its X and Y coordinates.

4.2 Abstract Syntax

The abstract syntax of FlowTeX is given in Fig. 4. Each expressible value has an associated node in the Abstract Syntax Tree. Specifically, there is a node each for the whole program, a class, a class attribute, a class method, a method parameter, each type of class relationship, a state, special state, state transition, and a position expression.

A design choice we took here was to keep position information separate from the items themselves, since they are unrelated to the functionality and semantics of the items. Hence, we do not include *X* and *Y* coordinates in the AST node of a class or state, but define a separate AST node to map items to their positions. This maintains the semantics of defining positions independently in the concrete syntax.

```
(define-datatype ast ast?
  [program (expression (list-of ast?))]
  [class (classname string?) (attributelist (list-of ast?)) (methodlist (list-of ast?))]
  [attribute (name string?) (datatype string?) (access-specifier access_specifier?)]
  [method (name string?) (returntype string?) (access-specifier access_specifier?) (paramlist (list-of ast?))]
  [param (name string?) (datatype string?)]
  [position (classname string?) (x number?) (y number?)]
  [associate (classname1 string?) (cardinality1 string?) (classname2 string?) (cardinality2 string?)]
  [inherit (childclass string?) (parentclass string?)]

  [state (statename string?) (label string?) (statelist (list-of ast?)) (transitionlist (list-of ast?))]
  [basicstate (statename string?) (statetype statetype?)]
  [transition (statename1 string?) (statename2 string?) (label string?)])
```

Figure 4: Abstract syntax and AST structure of FlowTeX.

5 IMPLEMENTATION DETAILS

FlowTeX consists of a command-line argument collector, a parser, an evaluator, and a file I/O mechanism to read concrete syntax as input and write output to files.

5.1 Parser

The parser takes a program written in the concrete syntax of FlowTeX as input, and outputs a single *program* AST node corresponding to the entire program. It is implemented using racket's inbuilt pattern matching. There is a separate matching clause for each expressible value, matched using the corresponding keyword. The parser is recursive in nature:

- At the topmost level, the parser loops over all expressible values in the program and recursively parses them. These values could be a class or a relationship. It then packs these parsed values into list and returns a *program* AST node containing this list of expressions.
- While parsing a class, the parser recursively calls itself on the class' *attributelist* and *methodlist*. It then packs these parsed lists (which are ASTs) into a *class* AST node and returns it.
- To parse an *attributelist*, the parser constructs three separate lists: one containing the access specifiers of all parameters, one containing the names of all parameters, and one containing the datatypes of all parameters. It then iterates over these lists simultaneously, packing them into a single list of *attributes*.
- To parse a *methodlist*, the parser constructs four separate lists: one containing the access specifiers of all methods, one containing the return type of all parameters, one containing the names of all methods, and one containing a list of parameters for each method. It first recurses to parse the parameter list of each method, and then combines all this data - by iterating simultaneously over all lists - into a list of *methods*.
- The parser matches method params using the *params* keyword. It constructs two lists: one containing the names of all parameters, and one containing the datatypes of all parameters. It then loops over these lists simultaneously and packs the data into a single list of the *param* AST type. This list is returned.
- The association relationship is matched using the *associate* keyword. There is no recursive parsing here; it directly returns an *associate* AST node.
- In the same way, the inherit relationship is a base case of recursion. It is matched by the *inherit* keyword. The parser returns an *inherit* AST node.
- While parsing a state, the parser recursively calls itself on the substates and their transitions. It then packs these ASTs into two separate lists of substates and subtransitions and returns a *state* AST node of the current state.
- A special state is parsed non-recursively after matching with the *basicstate* keyword in concrete syntax. A *basicstate* AST node is returned.
- A transition too does not require recursive parsing. It is matched using the *transition* keyword. The parser returns an *transition* AST node.

5.2 Evaluator

The FlowTeX evaluator takes a *program* AST as input and outputs *TikZ-UML* code for the expressed classes, relationships, states and transitions. The evaluator is recursive in nature, with the following rules:

- The evaluator iterates over all AST expressions contained in the top-level *program* AST node. Each expression is recursively evaluated into a string of the corresponding *TikZ-UML* syntax. For example, a *class* AST node will evaluate to a *umlclass* block in LaTeX, containing the class' attributes and methods. The evaluation results for all expressions, which are strings of syntax, are concatenated and returned, after adding some *TikZ-UML* boilerplate at the beginning and end of the final output string.
- To evaluate a class, the evaluator uses the following logic:

  - Recursively evaluate each attribute in the class' *attributelist*, obtaining a list of syntax-strings, one for each attribute.
  - Recursively evaluate each method in the class' *methodlist*, obtaining a list of syntax-strings, one for each method.
  - Concatenate the syntax-strings of all *attributes* into a single string and add required boilerplate around this super-string.
  - Concatenate the syntax-strings of all *methods* into a single string and add required boilerplate around this super-string.
  - Concatenate the above two super-strings, add required LaTeX boilerplate around the concatenated string (containing name and position information), and return the final output string as a fully-contained syntax-string of the class.
- To evaluate a class method, the evaluator uses the following logic:
  - Recursively evaluate each parameter in the method's *paramlist*, obtaining a list of syntax-strings, one for each parameter.
  - Concatenate the syntax-strings of all *params* into a single string, add required boilerplate around this super-string, and return it. This is a fully-contained syntax-string of the parameters of a class method.
- Evaluation of states is done in a similar way to classes:
  - Recursively evaluate each substate in the state's *statelist*, obtaining a list of syntax-strings, one for each state.
  - Recursively evaluate each transition in the class' *transitionlist*, obtaining a list of syntax-strings, one for each subtransition.
  - Concatenate the syntax-strings of all *states* into a single string.
  - Concatenate the syntax-strings of all *transitions* into a single string.
  - Concatenate the above two super-strings, add required LaTeX boilerplate around the concatenated string (containing name and position information), and return the final output string as a fully-contained syntax-string of the state.
- The evaluation of an *attribute*, *param* and both types of class relationships are not recursive. The same holds for a *basicstate* and state transitions. In each case, the evaluator packs the data into the required string syntax and returns the string as a self-contained unit, to be inserted within a larger string in the parent recursive call.

5.3 FlowTeX Pipeline

The complete pipeline followed by FlowTeX to convert an input program to output LaTeX code is given below:

- Read a program (written in FlowTeX's concrete syntax) from a specified file.
- Parse this program to obtain an Abstract Syntax Tree.
- Evaluate the generated AST to obtain an output string.
- Write the above output string into a specified file. This is valid LaTeX code which can be compiled and visualised in an editor or on the terminal, using a LaTeX compiler.

6 TESTING AND RESULTS

We have gone through multiple revisions of the concrete and abstract syntax of FlowTeX to make it simple and familiar. For example, to keep the concrete syntax similar to C and more natural, we changed method parameters to be defined in the order (access-specifier datatype name). However, they are stored in the AST in a different order, with variable name stored first due to its importance. The syntax also allows an empty *attributelist* or *methodlist* to be eliminated altogether, thereby reducing boilerplate. In a class, attributes can be defined before methods or vice versa. Similarly, in a state diagram, transitions need not be defined after states; any order works. Positioning too can be done anywhere in the program - before or after the item being positioned has been defined.

The FlowTeX parser has been tested on the following special cases:

- A class with no attributes.
- A class with no methods.
- A method with no parameters.
- A state with no substates.
- Invalid variable names (we enforce that the variable name must not be a number, but it can be anything else).
- Invalid access specifiers (we allow only three: public, private and protected).
- Invalid state types (we allow only two: initial and final).
- Attempting to position items which do not exist.
- All types of invalid syntax. For example, an inherit relationship with a missing parent class, a method with no return-type, or a transition with no "from" state.

To test the entire pipeline, we created a test suite. This involves writing FlowTeX code in files, running the FlowTeX interpreter on them to obtain output files, comparing these output files with ground truth output files (how the LaTeX code for the same UML diagram is supposed to look), and compiling FlowTeX's output in LaTeX to visualize the UML diagram. Results are shown in the following section.

Results for an example UML Class Diagram are shown in Fig. 5 (concrete syntax), Fig. 6 (output) and Fig. 7 (compiled output).

Results for an example UML State Diagram are shown in Fig. 8 (concrete syntax), Fig. 9 (output) and Fig. 10 (compiled output).

```
(flowtex
  (class test-class1)

  (class test-class2
    (attributelist (public int a) (private float b)))

  (class test-class3
    (attributelist (protected int a) (private float b))
    (methodlist (public double func1 (params (int p1) (bool p2)))
                (protected char func2 (params))))

  (associate (test-class1 test-class2) (0..1 *))

  (inherit test-class3 from test-class2)

  (position test-class2 0 -5)
  (position test-class3 10 -5))
```

Figure 5: Sample program to create a UML class diagram written in the concrete syntax of FlowTeX.

```
\begin{tikzpicture}
\umlclass[x=0,y=0]{test-class1}
{
}
{
}
\umlclass[x=0,y=-5]{test-class2}
{
+ a : int \\
-- b : float \\
}
{
}
\umlclass[x=10,y=-5]{test-class3}
{
\# a : int \\
-- b : float \\
}
{
+ func1(p1 : int, p2 : bool) : double \\
\# func2() : char \\
}
\umlassoc[mult1=0..1, mult2=0]{test-class1}{test-class2}
\umlinherit{test-class3}{test-class2}
\end{tikzpicture}
```

Figure 6: *TikZ-UML* code generated by FlowTeX corresponding to Fig. 5.

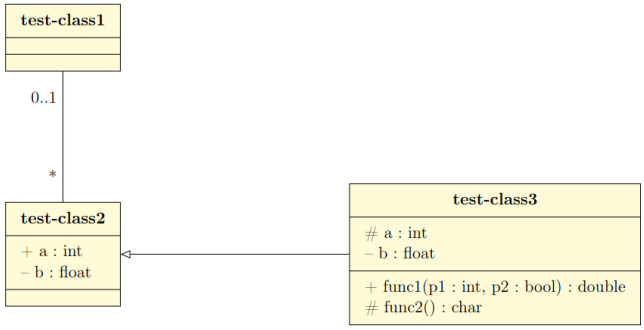

Figure 7: Generated UML class diagram as graphical output after compilation of FlowTeX's output in Fig. 6.

7 CONCLUSION AND FUTURE WORK

FlowTeX provides a convenient way to create UML class and state diagrams through code, significantly reducing time spent on these tasks compared to no-code solutions and general-purpose libraries such as *TikZ*. FlowTeX specifically targets domain of creating diagrams which, at the abstract level, are made up of "states" (these could be classes, states, graph nodes, etc.) and "transitions" (these could be relationships, state transitions, graph edges, and so on). FlowTeX's concrete syntax expresses these concepts in a purely functional manner, slightly ironical since the functional semantics of FlowTeX also represent Object Oriented concepts of classes, association and inheritance. Our parser utilises pattern matching to its full extent, making good use of our racket-like concrete syntax.

Due to the modular nature of UML class and state diagrams, FlowTeX can easily be extended to support any modular diagram system, such as graphs and trees, UML Use Case and Sequence diagrams, and automata theory diagrams. An interesting use case is extending the system to visualize Abstract Syntax Trees of programs to analyze control flow and variable scope. There is significant scope for future work in these extensions.

Another possible extension is adding support to include particular *TikZ-UML* features like color and font style directly into the output of FlowTeX as a post-evaluation step, allowing for greater customization through integrating FlowTeX with *TikZ-UML*'s existing features. This is a simple string concatenation problem.

```
(flowtex
  (state Amain "Global State"

    (state Bgraph "Sub Graph"
      (basicstate Binit initial)
      (state test1 "Test1")
      (state test2 "Test2")
      (basicstate Bfinal final)

      (transition test1 to test2 "Move")
      (transition Binit to test1 "Initialize")
      (transition test2 to Bfinal "End"))

  (basicstate Ainit initial)
  (basicstate Afinal final)
  (state visu "Visualization"))

  (transition Ainit to visu "Search")
  (transition test2 to visu "")
  (transition visu to Afinal "Close")

  (position test1 0 -4)
  (position test2 5 -4)
  (position Bfinal 5 -8)
  (position Ainit 10 0)
  (position Afinal 10 -8)
  (position visu 10 -4))
```

Figure 8: Sample program to create a UML state diagram written in the concrete syntax of FlowTeX.

```
\begin{tikzpicture}
\begin{umlstate}[x=0,y=0,name=Amain]{Global State}
\begin{umlstate}[x=0,y=0,name=Bgraph]{Sub Graph}
\umlstateinitial[x=0,y=0,name=Binit]
\umlbasicstate[x=0,y=-4,name=test1]{Test1}
\umlbasicstate[x=5,y=-4,name=test2]{Test2}
\umlstatefinal[x=5,y=-8,name=Bfinal]
\umltrans[arg=Move]{test1}{test2}
\umltrans[arg=Initialize]{Binit}{test1}
\umltrans[arg=End]{test2}{Bfinal}
\end{umlstate}
\umlstateinitial[x=10,y=0,name=Ainit]
\umlstatefinal[x=10,y=-8,name=Afinal]
\umlbasicstate[x=10,y=-4,name=visu]{Visualization}
\end{umlstate}
\umltrans[arg=Search]{Ainit}{visu}
\umltrans[arg=]{test2}{visu}
\umltrans[arg=Close]{visu}{Afinal}
\end{tikzpicture}
```

Figure 9: *TikZ-UML* code generated by FlowTeX corresponding to Fig. 8.

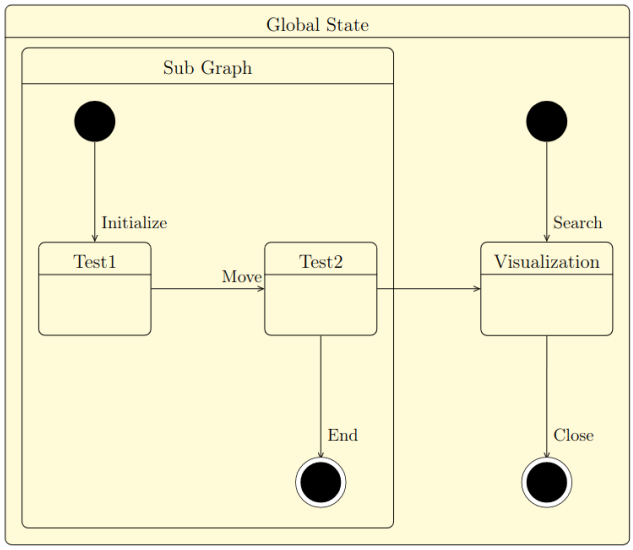

Figure 10: Generated UML state diagram as graphical output after compilation of FlowTeX's output in Fig. 9.

REFERENCES

[1] Nicolas Kielbasiewicz, "The TikZ-UML Package", March 29, 2016.
[2] LucidChart UML Diagram Tool, https://www.lucidchart.com/pages/examples/uml_diagram_tool.
[3] Creately UML Diagram Tool, https://creately.com/lp/uml-diagram-tool/.
[4] "Drawing UML with PlantUML", *PlantUML Language Reference Guide*, Version 1.2021.2.
[5] "UML Class Diagram Tutorial", *Visual Paradigm*, https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/.