Flowgen: Flowchart-Based Documentation Framework for C++

David A. Kosower
Institut de Physique Théorique,
CEA–Saclay, F–91191
Gif-sur-Yvette cedex, France
Email: david.kosower@cea.fr

Juan J. Lopez-Villarejo
Institut de Physique Théorique,
CEA–Saclay, F–91191
Gif-sur-Yvette cedex, France
Email: jj.lopezvillarejo@cern.ch

Serguei Roubtsov
Eindhoven University of Technology
Den Dolech 2, 5612 AZ
Eindhoven, the Netherlands
Email: s.roubtsov@tue.nl

Abstract—We present the Flowgen framework, which generates flowcharts from annotated C++ source code. It generates a set of interconnected high-level UML activity diagrams, one for each function or method in the C++ sources. It provides a simple and visual overview of complex algorithms, facilitating code development in international collaborations. Facilitating code development in international collaborations, one related closely to the algorithm as designed and written by its architects and developers, rather than a “low-level” one more closely tied to source code. Existing tools, described below, generate diagrams of the latter type.

We call the tool Flowgen. It generates a set of interconnected high-level UML activity diagrams, one for each annotated function or method in the C++ sources. Flowgen’s approach is independent of any particular programming style within the imperative paradigm [1]. Its approach is modeled on that of Doxygen [2], the de facto standard tool for generating documentation from annotated C++ sources. It binds source code and activity diagrams together, so that it is easier to maintain consistency between the two. It provides behavioral diagrams, which complement Doxygen’s structural information. We are currently applying Flowgen to the VINCIA code (http://vincia.hepforge.org), a plug-in to the high-energy physics event generator PYTHIA 8, used to simulate proton-proton collisions at CERN’s Large Hadron Collider. We believe it will be useful in a broad range of scientific computing codes.

Flowgen seeks to provide an easy-to-read, shared standard of communication (the activity diagrams) in a project, and thereby to promote the goals of:

- Facilitating code development in international collaborations, among developers with different levels of expertise and coding skills, possibly located in distant locations. This involves,
  - speeding up the training of new developers;
  - allowing the participation of “pure specialists” in the subject at hand, scientists with limited programming skills but an understanding of the high-level algorithm(s);
- providing better backwards traceability: a way to check that the final code package meets the scientific requirements;
- enabling iterative and incremental development of complex algorithms, a form of agile software development [4];

In this article, we present a proof-of-concept for these goals and address an additional one.
Increasing readability and transparency, displaying the flow of actions at a single glance.

In following sections, we discuss different aspects of Flowgen in more detail. In section II we describe related and similar tools. In section III, we present a simple example of annotations and the resulting output. In section IV, we describe the code annotations used by Flowgen; in section V, we describe how Flowgen is implemented; and in section VI, we discuss tests and lessons learned. We give some concluding remarks and an outlook in section VII.

II. RELATED WORK

Doxygen has emerged as a de facto standard for C++ structural documentation. It can generate either on-line documentation in HTML format or an off-line reference manual in LaTEX (or both) from a set of source files. In combination with the visualization tool Graphviz [5], it can generate class inheritance and call graphs. They contain structural information on how classes relate to each other, what class members they have, and (optionally) comments on what each class member’s role is. Annotations in the source code allow the programmer to enrich the documentation it produces. This way Doxygen makes it easier to keep the documentation consistent with the source code.

We adopt Doxygen’s philosophy of working with source files; Flowgen produces behavioral, high-level UML activity diagrams as a complement to Doxygen’s structural ones. They are intended to describe the semantics of what a code does, abstracted from C++ language-specific implementation details. They can cover these semantics at different levels of detail, at broad strokes corresponding to functions at the root of a call graph, or at a finer level corresponding to leaves of a call graph. They can also cover different levels of detail: a coarse level corresponding to long sequences of actions accomplishing a major task, as well as zooming in to a single action accomplishing an elementary task. Flowgen’s complementarity to Doxygen’s makes possible a future integration of the two tools.

A number of existing tools (both open-source and proprietary) allow programmers to generate activity diagrams from C++ source code. These include: Moritz [6] (an extension to Doxygen), IBM Rational Rhapsody [7], Crystal FLOW [8], AthTek Code to FlowChart Converter [9], Code Visual to Flowchart [10], AutoFlowchart [11], devFlowchart [12].

Although quite different in their abilities and maturity, all these tools have one common feature: they all generate diagrams based on the code, rather than on developers’ comments. Thus, the diagrams they produce are closely tied to the code and are low-level activity diagrams with implementation details shadowing the semantics of what the code actually does.

What makes Flowgen different from all those tools is its ability to combine high-level behavior described in annotations with implementation details parsed from the source code.

III. HOW FLOWGEN WORKS

As an example of using Flowgen, consider a simple set of annotated C++ source files: main.cpp, aux.h, and aux.cpp. They are shown in the following listings.

Listing 1: main.cpp
```cpp
#include "aux.h"
#include <iostream>

int main()
{
    int control_flag=0;
    // ask user whether to proceed
    std::cin >> control_flag;
    if (control_flag==1){
        // call shower
        // pointer to the object VINCIA
        VINCIA* vinciaOBJ = new VINCIA();
        vinciaOBJ->shower(); //$
    }
    return 0;
}
```

Listing 2: aux.h
```cpp
class VINCIA {
public:
    void shower();
};
```

Listing 3: aux.cpp
```cpp
#include "aux.h"
#include <iostream>

void VINCIA::shower(){
    // do VINCIA parton shower
    std::cout << "the parton shower code would go here";
    //@1) prepare system of partons
    //@2) do phase 1 of shower
    //@3)...
    return;
}
```

The comments marked with //@ are Flowgen annotations, which we shall describe in the next section. The tool uses them, along with extracted knowledge of the program’s control flow — decision points (if statements), loops, calls — to generate a single flowchart for each function or method.

Flowgen reads the source files of the project one by one and produces a set of interrelated .html files, connected via hyperlinks, which are stored in the folder flowdoc/. For this simple example, the output consists of the diagrams in Figure 1, which are included in the .html files. For code built using the make utility [13], it is easy to adapt the makefile to run Flowgen.

From the main.cpp source file, Flowgen generates main.html, containing a single diagram for the function main(), shown in the top diagram in Figure 1. Actions are the building blocks of the diagrams. They correspond to sets of statements in the code preceded by annotated lines, indicated by a leading //@. The if statement control structure, with condition control_flag=1, is picked up automatically and the flow paths are displayed in the diagram. An annotation using the //@ prefix at the end of a line of code serves to highlight the call to the function or method present on that line. In the example, the call to the method VINCIA::shower(), for which a separate diagram exists, is shown within the diagram.
Flowgen generates the aux.html file from aux.cpp. There is again a single diagram for the VINCIA::shower() method; but here, with two different zoom levels (0 and 1), shown in the pair of diagrams at the bottom of Figure 1. These zoom levels correspond to the numerical qualifiers following the ///$ annotations in listing 3 (no qualifier corresponds to '0').

In Figure 2, we give an example of a low-level activity diagram, for the main() function, that Flowgen may generate from code lacking annotations. This is roughly the kind of output generated by the tools mentioned in section II.

IV. CODE ANNOTATIONS

Flowgen produces high-level UML activity diagrams, which we’ll call simply activity diagrams, from annotated C++ code. It outputs these to a set of HTML files, one for each source code file.

The basic building blocks of an activity diagram are actions, each a statement or sequence of statements in the code. Each action conceptually accomplishes a discrete task. A sequence of actions builds up an activity. An activity may include different flow paths. An activity has a beginning and an end. In the diagrams produced by Flowgen, these are indicated by special round symbols (see the lower-left example in Figure 1). Conditional branches are indicated by diamond-shaped elements. The diagrams generated by Flowgen are interactive. In particular, they allow zooming and browsing. By zooming we mean the possibility of inspecting the graphical description at different levels of detail, as previously annotated by the programmer. By browsing we mean the possibility of navigating through the network of interconnected activity diagrams associated with different functions or methods in a package. Navigation is implemented using standard HTML hyperlinks.

In the code, activities correspond to annotated functions or methods. Flowgen’s annotation grammar recognizes Doxygen annotations of functions or methods. This feature will eventually allow their use as additional comments in Flowgen activity diagrams. The actions, along with the level of detail to which they correspond (zoom level), are specified in the source code by the programmer. The basic syntax is as follows,

///$ (options) action description

The $ symbol is part of the syntax which allows Flowgen to distinguish its own annotations from Doxygen ones.

The beginning and end of the full activity to which the action belongs are determined by analyzing the code itself, as are the different flow paths within the activity.

An up-to-date specification for the annotation can be found on the project’s website, http://jlopezvi.github.io/Flowgen. These include the formal specifications (Extended BackusNaur Form). Here we summarize the essentials:

```cpp
int class::activity_method(){
  int a;
  ///$ do something
  // we print using std::cout
  std::cout << "do 1" << endl;
  ///$ do other thing
  std::cout << "do 2" << endl;
  return 0;
}
```

Fig. 3: Example of actions: annotated code and corresponding graphical form in an activity diagram.

- Annotations describing actions, using the syntax given above, are the key added elements that allow Flowgen to generate a rich description. (See the example in Figure 3.) An annotation specifies what succeeding lines of code (up to the next annotation or an annotated flow-control structure) are doing. The added ‘$’ distinguishes an annotation from a regular C++ comment, allowing the programmer to choose explicitly what appears in the activity diagrams.
using namespace std;
void activity_function(int a){
int c=2;
if(a>0) {
    //§ action 1
    cout<<"do 1"<< endl;
    //§ [subcondition for true]
    if (a>c) {
        //§ action 4
        cout<<"do 4"<< endl;
    }
    //§ [subcondition for false]
    else if (a==-1) {
        //§ action 3
        cout<<"do 3"<< endl;
    }
    else {
        cout<<"do nothing"<< endl;
    }
    return;
}

Fig. 4: Example of nested if-statements: annotated code and graphical form in an activity diagram.

- In if-elseif-else statements (see Figure 4), annotation allows the controlling condition to be described in a human-readable way. The annotation is //§ [condition description], which should be placed on the line immediately preceding the if, elseif or else statement that it describes (square brackets are part of the syntax). Loop control structures (while, do-while, for) allow similar annotations.

... 
//§ last action
//§ [return value]
return xVar;
...

Fig. 5: Example of a return statement: annotated code and graphical form in an activity diagram.

- Annotations preceding return statements (see Figure 5) allow the return value to be specified in a human-readable way. (This feature has not yet been implemented.)

... 
//§ <parallel> action 1
code
//§ <parallel> action 2
code
//§ <parallel> action 3
code
...

Fig. 6: Example of parallel actions: annotated code and graphical form in an activity diagram.

- Parallel actions (see Figure 6): the tag '<parallel>' allows the programmer to indicate whether a sequence of actions could (in principle) be executed in parallel. (This tag has not yet been implemented.)

- A postfix annotation, code_line_with_a_function_call //§ allows the programmer to highlight calls to functions or methods (see Figure 1 and the call to the method VINCIAShower() in the example in the previous section). The call will appear explicitly in the diagram. This annotation also inserts a hyperlink to the diagrams for the functions or methods. This allows a developer to browse from the caller’s diagram to the called function’s diagram.

- Zoom levels: the programmer can indicate at which level of detail a description of an action should appear by adding an integer immediately after the opening '//§' of an annotation (see Figure 1 and the associated listing 3). Higher numbers indicate a finer level of detail; the zoom level is 0 by default, corresponding to the coarsest level of detail. This makes different zoom levels possible in visualizing the HTML output.

V. IMPLEMENTATION

In this section we discuss the implementation of Flowgen. In the first of two subsections, we discuss the requirements arising from the specifications presented in the previous section, as well as the choice of technologies; in the second, the design concept and the specific implementation.

A. Requirements and Technologies

We can classify the annotations discussed in the previous section into three groups from a ‘technical’ point of view. This classification is useful to understand the requirements of our tool.

- actions (//§ (options) action description): these will be given sequentially within compound statements, that is sequences of code lines inside braces. Action annotations can appear anywhere inside a compound statement.
– contextual annotations adding descriptions to control-flow structures
  \texttt{//\$/description): the precise position of these annotations is important, and they must be associated to the corresponding control structure (\texttt{if}, \texttt{for}, etc.)
– call highlighters (\texttt{code\_line //$/}): the preceding code line must be analyzed to identify function calls; and the calls should be matched with the corresponding diagrams for the called functions, if they exist.

The need to link actions to the C++ code, and in particular with detailed knowledge of the syntax of the C++ code, makes it necessary to use a C++ parser. We have used the \texttt{libclang} library of the Clang project (http://clang.llvm.org/). We use it to perform the syntax analysis phase of C++ parsing, which yields an abstract syntax tree (AST). This tree is then used by Flowgen to extract information it needs.

Clang is a C language family front-end for the LLVM compiler\textsuperscript{1}. Clang’s development is completely open-source, with several major software development companies involved, including Google and Apple. Clang features static analysis utilities and bindings to Python via a standardized library called \texttt{libclang}. Clang also includes full support for annotations with the Doxygen format (called “full comments”) but, at present it leaves remaining comments out of the generated AST. The documentation is at present mostly at the developer level.

For most of our purposes, however, regular expression and scripting techniques are well-suited and convenient. For these, our language of choice is Python (more specifically, Python 3).

As far as activity diagram visualization concerns, there are at least two free solutions that automate the generation of graphs: UMLet [14] (http://www.umlet.com/) and PlantUML (http://plantuml.sourceforge.net/). The programs draw diagrams from a description given in textual form in a simple and intuitive language. We have chosen to use PlantUML; we hope that its continued development will also enhance the capabilities of our documentation tool. As an interface to a visualization system, Flowgen uses standard HTML. This choice allows the use of any web browser as the visualization system.

We end this subsection by listing the software prerequisites for Flowgen:

\begin{itemize}
  \item LLVM-Clang 3.4 (or later) + Python3 bindings
    http://clang.llvm.org/get\_started.html
    https://github.com/kennytm/clang-cindex-python3
  \item Python3
    http://www.python.org/getit/
  \item PlantUML (included in the Flowgen distribution)
    http://plantuml.sourceforge.net/
\end{itemize}

B. Design Concept and Implementation

In order to produce activity diagrams, Flowgen must execute the following four steps:

1) \texttt{Read sources} (annotated C++ code), parse them, and link the parse tree to the annotations, using Clang and Python3.
2) \texttt{Produce an abstract representation of the diagram}, using Python3.
3) \texttt{Render the abstract representation of the diagram into graphical form}, making the diagram layout (widths, lengths, fonts, colors, etc.) This step uses PlantUML.
4) \texttt{Embed the generated set of diagrams into HTML files}, to allow zooming and browsing, as explained in sect. IV. This step uses Python3.

We present a more detailed account of how Flowgen operates in Figure 7. We distinguish three phases, which however do not precisely correspond to the list above.

In the initial ‘build\_db’ phase, for each source file (headers excluded), a database is generated which contains a list of the annotated functions or methods. Generated database files are text files and carry the extension ‘.flowdb’. This phase is necessary for multi-file projects, because Clang cannot simultaneously process multiple translation units.

A Python script controls the main phase, ‘makeflows’. It first reads the sources and calls Clang to get information on the annotated functions or methods: namely, their starting and ending locations in the source files. The script then processes the corresponding ranges line-by-line. Some annotations (actions) are identified by simple regular expression parsing. More complicated structures are captured by using the Clang library. For each source file, the script writes a corresponding text file (with extension ‘.txt’) containing a PlantUML description, giving the commands to draw the diagrams for

\begin{figure}
\centering
\includegraphics[width=\textwidth]{flowgen_flowchart}
\caption{Flowgen’s flowchart, with the three required tools and libraries: Python, Clang and PlantUML}
\end{figure}
all the annotated functions or methods. PlantUML is then run
(externally) in order to obtain the diagrams in PNG format,
as well as image maps in CMAPX format. The latter are used
in the HTML pages to attach hyperlinks to certain rectangular
regions of the PNG images (for example, to attach hyperlinks
to calls to functions or methods).

Finally, in the ‘makehtml’ phase, another Python script
generates an HTML file for each source file. The HTML files
include the PNG images and use the information in the CMAPX
files.

The three phases are automated in a makefile.

VI. FLOWGEN EVALUATION AND DISCUSSION

We have tested our initial implementation of Flowgen
on a variety of source files, which include code with nested
if statements, calls to functions and class methods, annotation
with different zoom levels, and links to be followed in
browsing. A full, realistic example for a single method in the
VINCIA code mentioned in the introduction can be found on
the project’s website (http://jlopezvi.github.io/flowgen/). The
example is a long procedural method where separation into
several smaller methods is possible and may be desirable.
The code was taken from another VINCIA developer. We
selected amongst those comments already present the ones
that reflected a description of the actions performed by the
code, and annotated them, including a zoom level where
appropriate. We also indicated where parallel processing was
possible. In addition, we annotated some conditions for if
statements and loops. We believe that the resulting diagram
makes understanding the algorithm much easier, and that
this understanding compensates for the additional effort in
annotation.

We believe that even the elementary example depicted in
Figure 1 shows the benefits of the tool we are proposing.
The diagrams combine two different views of the code, a
high-level semantic view on the one hand, with code-level
implementation details such as branching, or variable and
method names important to annotated activities. It offers a
common ground for specialists of different backgrounds to
collaborate more efficiently.

We regard the present implementation as a proof of concept.
We note in passing that the activity diagrams generated
from the code by Flowgen can be modified by hand, as the
PlantUML input files are text files. This could in principle be
used to modify or evolve the design of the code; the code and
accompanying annotations could then be updated to match.
The Flowgen tool can thus be used to facilitate iterative and
incremental (‘agile’) development at a higher level than direct
coding. We intend to apply Flowgen more widely within the
VINCIA collaboration, and to refine it as we gain experience.
The present version of the tool is in any case available from
the project website.

VII. CONCLUSIONS AND OUTLOOK

We have described Flowgen, a documentation tool that
generates high-level UML activity diagrams from annotated
C++ sources. These diagrams give a description of the dy-
namic behavior of the code. Flowgen is able to combine
the high-level behavior view with the implementation details
parsed from the source code. The tool is complementary to
the Doxygen documentation tool, which provides the user
with structural information about static aspects of the code.
Flowgen employs annotations similar in spirit to those of
Doxygen, designed so as not to interfere with the annotations
used by the latter. As Flowgen matures, this preserves the
possibility of combining the two tools.

A behavioral description of a software package, using
activity diagrams, allows us to see at a glance its procedural
flow of actions. Flowgen gives a graphical representation of
this procedural flow, and adds two other capabilities: the
possibility of zooming to different levels of detail; and the
possibility of browsing to diagrams for other called functions
within the package.

Flowgen requires annotating the code to indicate the
discrete actions and select calls to hyperlink, and optionally
to add descriptions to control structures, indicate parallelizable
code, and different levels of detail for later visualization. The
additional effort to produce a basic visualization is modest;
a complete high-level description would obviously require
additional effort in rethinking the textual parts of comments.

We have designed the tool primarily for code written in a
procedural style within the imperative programming paradigm
[1], the one which encompasses the bulk of scientific code. It
is primarily designed for developers and designers, rather than
users, but is explicitly intended to address a broad spectrum of
programming abilities, from skilled programmers to designers
with an understanding of the underlying science and algorithms
but limited programming abilities. Because we have relied in
general principles for its design, we believe that Flowgen
can be used for general scientific computing packages.

REFERENCES

[1] Pratt, Terrence W. and Marvin V. Zelkowtiz. Programming Languages:
Hall, 1996.
Junger, P. Mutzel (Eds.), Graph Drawing Software, Springer-Verlag,
products/en/ratirhapfami
CrystalFLOW.htm
debug-to-flowchart.html#.U5chEvmSzNh
mak e/
User Interface Setting. Proceedings of 11th International Conference on
Enterprise Information Systems (ICEIS 2009), Milan, May, 2009;
M. Auer, T. Schuurtschenthaler, S. Biffl, Proceedings of EUROMICRO