STATECHART VIEWER

- Summer of Code project -

Organization: *Boost*

Mentor: *Andreas Huber*

Student: *Ioana Tibuleac*

PROJECT GOAL

The goal of this project is to create a tool that enables the Boost.Statechart users to visualize finite state machines.

SUMARY:

1. USER'S GUIDE

1.1 Requirements

The application requires the following to be installed:

- a Linux operating system
- the g++ compiler (version 3.4.4 or newer)
- the GCC-XML tool (latest CVS snapshot)
- the Graphviz tool (1.16 or newer)
- the Boost library (the BGL and Statechart libraries)

1.2 Compilation

In the application's root directory run:

make

This will create an executable file named **viewer.**

1.3 Usage

In a console run:

./viewer source_files output

source_files = specifies the source files (files that contain the definitions of the finite state machines)

output = specifies the name of the .png file containing the output diagram.

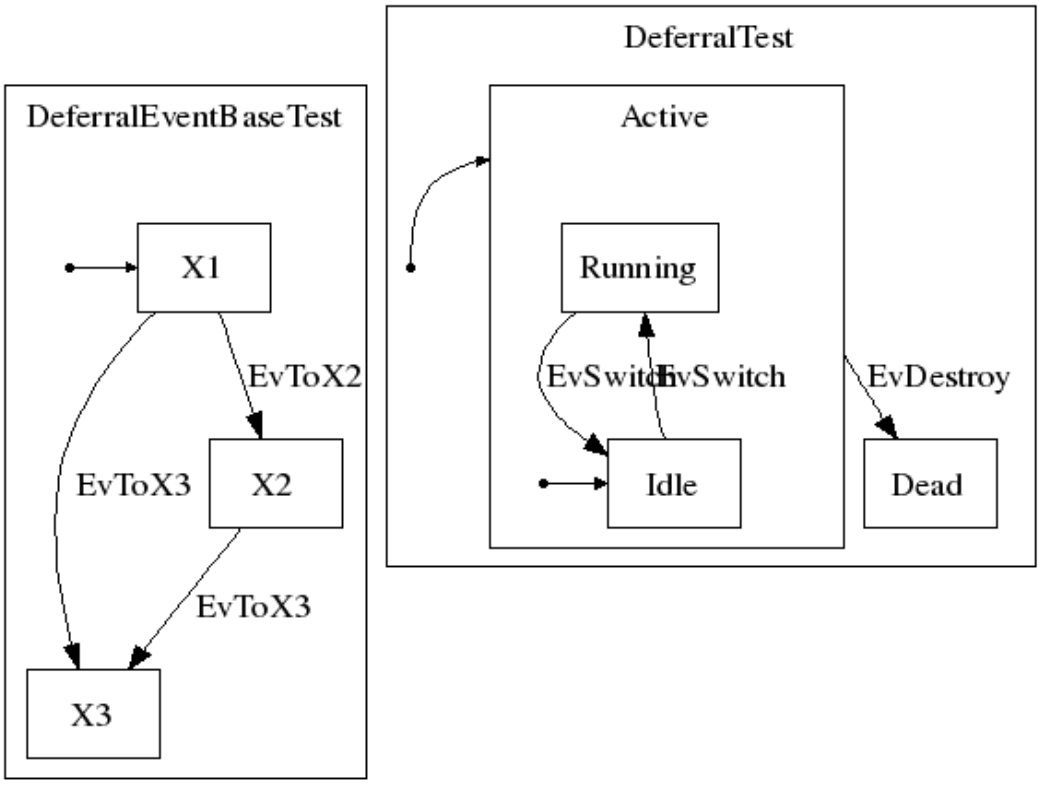
A run of the program outputs:

- one png file containing a state machine diagram
- one file containing the diagram in the Graphviz dot language

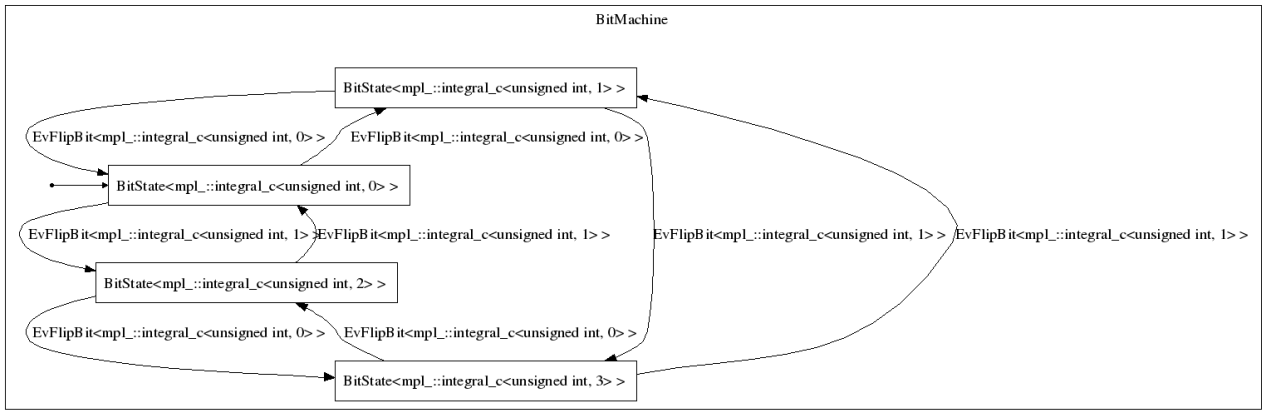
1.4 Examples

Following are the results of running the application on some of the examples defined in the boost/statechart CVS tree.

DeferralTest.cpp



BitMachine.cpp



2. TOOLS

2.1 GCC-XML

http://www.gccxml.org/HTML/Index.html

- What does it do?

  GCC-XML generates an XML description of a C++ program from GCC's internal representation.

- How does the Statechart Viewer uses it?

  The Statechart Viewer application uses GCC-XML on the source files indicated by the user; the output is an .xml file which will be parsed in order to extract useful information regarding the FSMs. This information includes: the state machines defined by the user, the states contained by each machine, transitions between states and the events that trigger them.
  The XML file generated by GCC-XML is quite large (~ tens of thousands of lines). This is because it contains not only relevant data (extracted from the processed source file) but also noise data (from the included headers and compiler generated structures).

- License:

  The GCC-XML Copyright is an open-source, Berkley-style license. It allows unrestricted use, including use in commercial products.

2.2 irrXML

http://xml.irrlicht3d.org/

- What does it do?

  Irrxml is a very lightweight XML parser for C++. It provides forward-only, read-only access to a stream of non validated XML data. Some of its features include:

  it is fast and has very low memory usage
  it is platform independent and works with lots of compilers.
  it has no external dependencies, it does not even need the STL.
  it is very small: it only consists of 60 KB of code
  it provides a very intuitive interface and it is easy to build.

- How does the Statechart Viewer uses it?

  IrrXML is used in the Statechart Viewer application to parse the XML output by the GCC-XML build of the source files containing the state machine definitions.
  Due to the nature of the XML file we're dealing with (great number of lines and containing generated data - machine readable), the obvious choice for a parser is a SAX based one. All the features of irrXML recommend it as the best choice.

2.3 Graphviz

http://www.graphviz.org/

- What does it do?

  Graphviz is open source graph visualization software. It consists of a package of open source tools initiated by AT&T Research Labs that render graphs specified in DOT Language scripts. It also provides libraries for software applications to use the tools. Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes.

  Dot is a command line tool that draws directed graphs. It reads graph text files and writes drawings either as graph files or in a graphic format (gif, png, svg, postscript).

- How does the Statechart Viewer uses it?

  The application generates a dot script. The command line tool – **dot**, will then be run on this script. This will output one png file containing the representation of the specified state machine.

- License

  Graphviz is free software licensed under the Common Public License.

3. ARCHITECTURE

This section presents an overview of Statechart Viewer's architecture. This is described in terms of highlighting the major development milestones.

3.1 Development milestones:

- **Phase 1**

  use the GCC-XML tool to transform the code implementing the state machine – the .cpp and .hpp(s) supplied by the user, to an XML file.

  parse the XML file obtained at the previous step and extract the relevant information; this steps involves only extracting certain tags from the XML file, the tags that define the states, the transitions and the events that trigger transitions.

- **Phase 2**

  the tags extracted during the previous phase are now parsed in order to obtain meaningful information from the raw XML strings; this information will then be used to build corresponding objects for each state machine, state and transition.

- **Phase 3**

  use the lists of states and transitions obtained during phase 2, to build a BGL graph.

  use the write_graphviz() function provided by BGL to obtain the dot script corresponding to the graph.

  run the command line tool dot, provided by Graphviz, to obtain one .png file representing the state machines defined by the user in the specified source files.

For more informations on each phase see below.

3.2 Phase 1

The current version of the application accepts only FSMs defined in one .cpp file and one or more .hpp files.

3.2.1 The XML file format:

Following are the relevant tags and attributes:

*<**File** id = ... name = ...>* => specifies that this file has been analyzed and included in the output by GCC_XML

- **id** = the id that GCC_XML has associated with this file
- **name** = the name of the file

*<**Struct** id = ... name = ... file = ... members = ... bases = ...>*

- **id** = the id of the structure
- **name** = the name of the structure
- **file** = the file in which it was defined
- **members** and **bases** = lists of ids of tags that are relevant to this structure's definition

*<**Class** id = ... name = ... file = ...>*

same as for **Struct** tag

*<**Typedef** id = ... name = ... type = ...>*

- **id** and **name** – as before (in Statechart Viewer's case only Typedefs with name = “reactions” are relevant)
- **type** = points to the Class tag corresponding to this Typedef

3.2.2 Parsing

Three passes of the XML are necessary due to the unfortunate order of tags and the nature of the parser (SAX – forward only).

- **Pass 1**

  identifies and stores the file IDs for the source files indicated by the user and the file IDs for the relevant boost headers:

  - statechart/state_machine.hpp
  - statechart/asynchronous_state_machine.hpp
  - statechart/simple_state.hpp
  - statechart/state.hpp
  - statechart/transition.hpp
  - mpl/aux_/preprocessed/gcc/list.hpp
  - mpl/list/aux_/item.hpp

  How? By finding all **File** tags with the respective *name* attribute and store the *id* attribute. These tags are located at the end of the XML file so additional passes of the file are necessary.

- **Pass 2**

  identifies the user defined structures (states and location of transitions)

  How?

  - Find all **Struct** tags that have the *file* attribute equal to one of the source file ids. Store for each such tag the *name, members* and *bases* attributes.
  - These tags are located at the beginning of the xml so parsing can continue; identify the **Class** tags with *id* equal to the first id in the *bases* list. If the *file* attribute indicates that the structure was defined in simple_state.hpp or state.hpp it means we found a state => store the *name* attribute (it contains the raw definition of the state). Example of a raw state

definition:

```
"simple_state<Running,Active,boost::mpl::list<One,Two,
mpl_::na,mpl_::na,mpl_::na,mpl_::na,mpl_::na,mpl_::na,
mpl_::na,mpl_::na,mpl_::na,mpl_::na,mpl_::na,mpl_::na,
mpl_::na,mpl_::na,mpl_::na,mpl_::na,mpl_::na,mpl_::na>,has_no_histor
y>"
```

- For each of the IDs contained in the *members* id list of the **Struct** tag from above, check if the corresponding tag is of type **Typedef** with *name* = “reactions”. If so, we have found a transition (it is identified by the *type* attribute of the **Typedef** tag). This tag is usually somewhere before that point so an additional pass is again necessary.

- **Pass 3**

  obtains the raw definitions of all transitions

  How? By finding the tags with the IDs identified durin the last step of the second pass; when such a tag in encountered, store the *name* attribute.

3.3 Phase 2

During phase one we obtained lists of unparsed states and unparsed transitions. This phase consists of parsing those raw definitions and building the appropriate objects.

The objects and some of their members:

- **state_info**

  name = indicated the name of the state

  context = indicates the location of this state in the statechart

  inner_states = if the state is hierarchic this is a list of all the states that it directly contains

  is_initial = this specifies whether the state is or isn't initial within its context

- **transition_info**

  initial – the source state of the transition

  final – the destinatn state

event - the trigger

The state_info objects are arranged into a tree structure (context is the parent node and inner_states are the children).

The transition_info objects are independent from one another; they will be stored in a vector.

3.4 Phase 3

This phase consists of building a BGL graph from the state_info and transition_info objects constructed above. Each state_info object maps to a vertex in the graph and each transition maps to an edge.

The process of initializing the vertices of the graph is a recursive one. It starts with the root node, which is the state machine and traverses the tree structure of state_info (by traversing the inner_states for each state_info). Each vertex will get the apropriate Graphviz attributes (label, shape, style). Nodes that contain other nodes are clusters; for each cluster a new subgraph will be added.

The process of initializing the edges is quite simple; it consists of iterating through the transitions vector and just use the initial's and final's vertex descriptors to connect the states by an add_edge call.

Graphviz does not treat clusters as regular nodes. So in order to be able to treat clusters like simple nodes (for example to add an edge directly coming out of or going into a cluster) I have created an invisible node inside each cluster. These invisible nodes will act as an alias for the cluster.

5. SHORTCOMINGS. FUTURE FEATURES

The current version has the following *shortcomings*:

- The only reactions shown are transitions. Custom reactions cannot be represented due to a GCC-XML shortcoming: it does not dump function bodies. Information essential to a custom reaction (the destination source of the transition) is found in the body of the **react** function.

- Hierarchic states can only be represented as rectangles. This is due to Graphviz: it does not allow other shapes (besides rectangle) for clusters of nodes. In order to maintain consistency all the other nodes are box shaped too.

- Orthogonal regions are not separated by dashed lines. This is possible only for simple states (not containing inner states) by setting their shape to **record**. This is not possible with complex states, because Graphviz does not allow clusters of nodes to be treated as regular nodes. A cluster of nodes cannot have a **record** shape.

- Self transitions for hierarchic states are not possible. The solution was to add one unnamed state for each self-transition and draw an edge from the cluster to the unnamed state (with the event attached to it) and one from the unnamed state back to the cluster (with no event associated).

- In-state transitions are not possible. Graphviz does not allow the head of an edge to be contained in the tail of the same edge.

Future features:

- represent diagrams of finite state machines contained in more than one .cpp file

- represent orthogonal states separated by dashed lines