Discover C++ applications via static code analysis.

A concrete example

Wolfgang Spahn

2022-12-14

When complexity is beyond basic and documentation is pure, discovering existing C++ frameworks can be difficult. This is where automatic mapping of C++ entities into knowledge frames (Minski's classic AI approach to knowledge representation [1]) via static code analysis can help. To show the benefit, let's discover the C++ lib OGDF, a powerful open graph drawing framework[2]. With its 500kLOCs of header and source lines, it is already a fairly complex framework. Failure to initialize elements will cause runtime errors. So usage is not obvious, and good documentation, especially on what needs what, and on do and don't, would help.

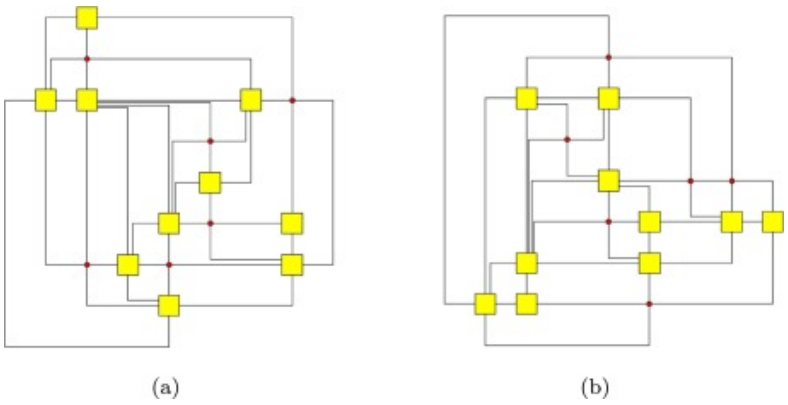

Figure 1: OGDF planarization

In this particular case, We have a getting-started-guide, examples, and papers addressing the scientific background. And of course a doxygen run, which collects doc strings. But it leaves quite a lot of areas open. Quite a few doc-strings are just repeating what has been already said in the class or function name. That's not nice but it reflects daily live of C++ programmers - in companies and academics.

What can we do to get started in an efficient way? Let's harvest implicit knowledge.

1 Build a knowledge graph

Even when not written down, the knowledge is implicitly there. Code has been created by humans, which need to think in language constructs. We postulate that C++ constructs must in some way refer to language ones. Where are the domain names of context, modules, concepts, entities, objects, instances, rules, relations, attributes etc.. How to they relate, in what way? And in each category there will be some hierarchy as otherwise no human could have developed the framework. But how to make this a practical approach?

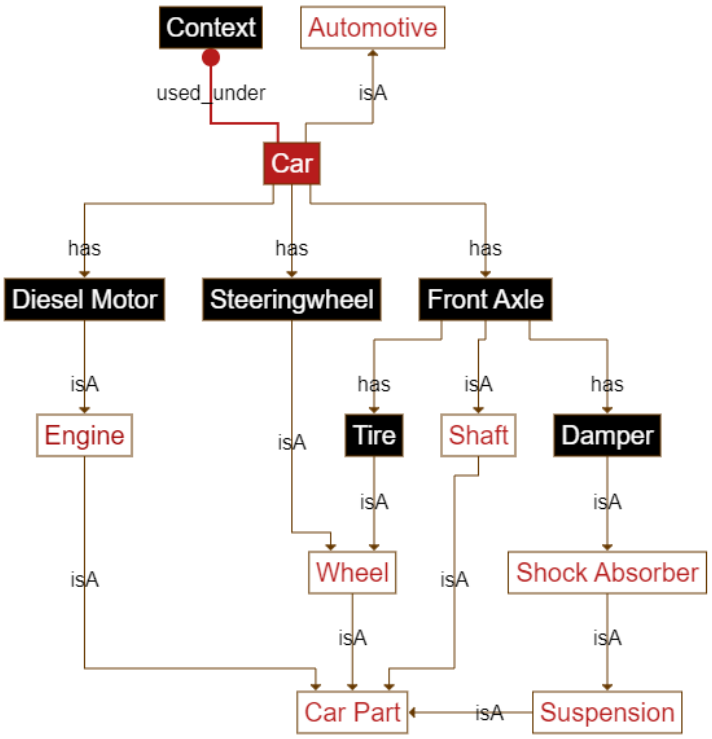

Figure 2: A knowledge graph of a car

To avoid to open the full complexity of knowledge representation and natural language understanding, we start with a quite simple strategy. Map C++ to just a few natural language constructs, which are used to collect knowledge frames in classical AI (f.Ex. concept net or wordnet) and store them in a knowledge graph.

- Parse the Code to towards its AST representation.
- Select an entry point, which represents the frame
- Map C++ constructs to natural language ones.

In a sense, we create a knowledge frame or archetype from the C++ code, which should describe the domain knowledge implicitly present in the code. We get a graph showing the base concepts (framed in red), the concepts we need to select for implementation (filled in black), the entry point (filled in red), the initial call call(GA), and their `isA` and `has_m_X` relations. The graph produces an implementation space of 20-50 header files in a fairly compact way. It is important to note that it does not just cover one, no it covers a full family of behaviors.

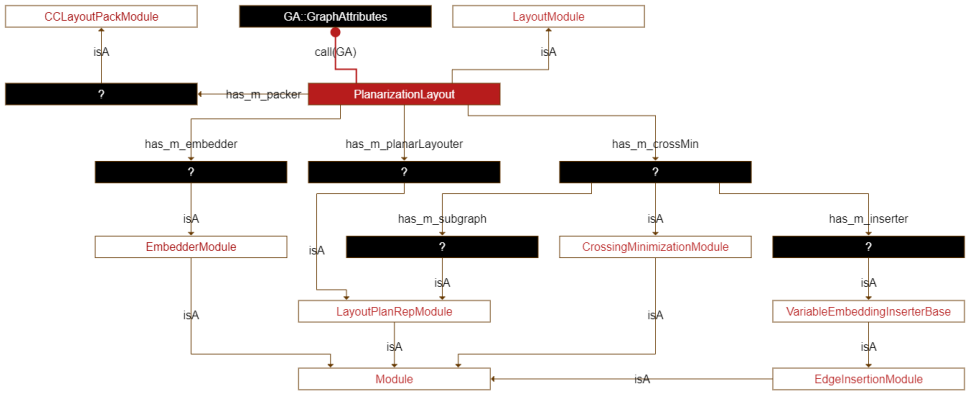

Figure 3: Result of static code analysis and the mapping C++ constructs into NLP concepts and relations, to get a big picture.

2 Learn from the Graph

The graph is quite compact. In AI talk it's an archetype and covers all layout variants you can implement with `PlanarizationLayout`.

There are three fundamental concepts and 7 base concepts

- `Module`: Base concept for all algorithms implemented
- `CCLayoutPackModule`: Packs connected components (CC) in a layout engine
- `LayoutModule` : Base concept of all e2e engines implemented
- `CrossingMinimizationModule`: Minimizes crossing of edges of non planar graph
- `EdgeInsertionModule`: Inserts edges in an optimized way
- `VariableEmbeddingInserterBase` (inherits `EdgeInsertionModule`)
- `EmbedderModule`: Algorithms which embeds graphs in some way
- `LayoutPlanRepModule`: Algorithms which calculate layouts of planar graphs

By following the `has_m_X` dependencies we can envision how to implement a `PlanarizationLayout`:

- construct the entry class `myPlanaritationLayout` and assure that
- all `has_m_X` concepts are implemented, which are given by
- the base classes defined by the `isA` hierarchy shown in the graph
- all attributes are set (not shown) and finally execute
- `myPlanaritationLayout.call(GA)`

Let's compare this with a real code example. The layout call is embedded in a GraphIO context that loads a sample graph data and writes the layout to disk, which is another use case we could discover by using `GraphIO` as an entry point.

Looking at the actual construction of `PlanerizationLayout`, we can confirm our knowledge graph. It describes the creation of all concrete instances of the base classes we extracted. Some attributes are overridden, others are used as defaults. Finally, `pl.call(GA)` is called. Our graph has captured the main concepts needed to write orthogonal planarization layout code in one view.

```
using namespace ogdf;

int main()
{
    Graph G;
    GraphAttributes GA(G,
      GraphAttributes::nodeGraphics | GraphAttributes::nodeType |
```

```
      GraphAttributes::edgeGraphics | GraphAttributes::edgeType);

    if (!GraphIO::read(GA, G, "ERDiagram.gml", GraphIO::readGML)) {
        std::cerr << "Could not read ERDiagram.gml" << std::endl;
        return 1;
    }

    for (node v : G.nodes)
    {
        GA.width(v) /= 2;
        GA.height(v) /= 2;
    }

    PlanarizationLayout pl;

    SubgraphPlanarizer *crossMin = new SubgraphPlanarizer;
    PlanarSubgraphFast<int> *ps = new PlanarSubgraphFast<int>;
    ps->runs(100);
    VariableEmbeddingInserter *ves = new VariableEmbeddingInserter;
    ves->removeReinsert(RemoveReinsertType::All);

    crossMin->setSubgraph(ps);
    crossMin->setInserter(ves);
    pl.setCrossMin(crossMin);

    EmbedderMinDepthMaxFaceLayers *emb = new EmbedderMinDepthMaxFaceLayers;
    pl.setEmbedder(emb);

    OrthoLayout *ol = new OrthoLayout;
    ol->separation(20.0);
    ol->cOverhang(0.4);
    pl.setPlanarLayouter(ol);

    pl.call(GA);

    GraphIO::write(GA, "output-ERDiagram.gml", GraphIO::writeGML);
    GraphIO::write(GA, "output-ERDiagram.svg", GraphIO::drawSVG);

    return 0;
}
```

In the example above, 5 concepts have been derived from a base that is represented by 1 slot in the archetype.

- `OrthoLayout:LayoutPlanRepModule`
- `EmbedderMinDepthMaxFaceLayers:EmbedderModule`
- `SubgraphPlanarizer:CrossingMinimizationModule`
- `PlanarSubgraphFast:LayoutPlanRepModule`
- `VariableImbeddingInserter:VariableImbeddingInserterBase`

Which gives a filled frame:

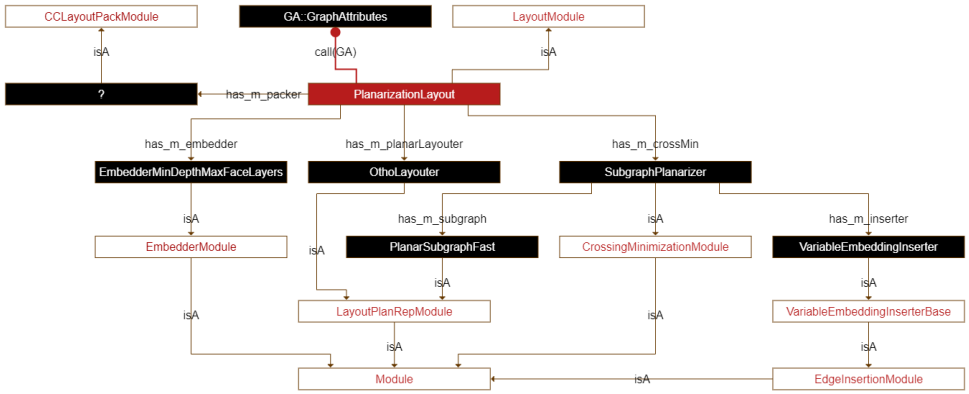

Figure 4: When comparing with a code example we can fill the frame and get our basic assumptions confirmed.

As expected one slot is not filled, as it is dealing with connected components, which is not applicable to our graph type (no components used).

We saw just one way to fill the class inheritance for the empty slots. As you might guess, the archetype can be filled in quite different ways. It presents the space of all potential implementations. Some are intended, others are just misuse (which will fail during runtime). To make this explicitly written down is another obvious benefit.

3 Conclusion

Fur sure it's not always as simple to recover main concepts in C++ code. Typically you have to map more constructs of C++. Friend would be a next step. And you will see more complex dependencies. But the `isA`, `hasX` hierarchy is always a good starting point. It gives a dense knowledge graphs of concepts, covering a wide area of code.

By comparing the resulting concepts with usage examples, the knowledge graph can be verified. Valuable explicit knowledge can be created. We get an initial understanding of the code from which we can proceed.

I think this is a good way to develop trust in the comprehension, and to create top level documentation automatically, which assures proper usage of the framework.

Via this we have a fast path to build much more comprehensive documentation/understanding from code not only in the graph drawing world.

References

[1] M. Minsky, "Minsky's frame system theory," in *Theoretical issues in natural language processing*, 1975.

[2] "OGDF – Open Graph Drawing Framework — ogdf.uos.de." https://ogdf.uos.de/.