Returning data-flow to asynchronous programming

Matt Gilbert, Senior Staff Engineer, Qualcomm Technologies, Inc. 2018-04-16

Background

- Hardware design focuses on information flow (data and control): how do you compose the pieces of execution to balance speed, efficiency, and area?
- We self-impose asynchronicity to avoid accidental time travel between hardware timing boundaries.
- HW execution is modeled as concurrent asynchronous events, using publish/subscribe as the fundamental building block of composition (distributed state, à la actors).

The problem

Applications composed of decoupled components, connected at runtime, lead to a callback nightmare -- e.g. how do you statically follow data-flow through the system?

Our solution

Use the static information we have to reconstruct the decoupled callgraph.

Publish/subscribe library runtime connections are based on static information:

- Connections are type safe.
- Strong emphasis on connecting events to state transitions means little usage of dynamic string creation.

Realization: we have enough static information to re-create a version, or multiple versions of the dynamic data-flow.

Basics

publish/subscribe communication is connected through a Registrar of connections.

Consumer:

```
//          signature           name
//              ↓                 ↓
reg.lookup<void(std::string)>("print channel").hook([] (const std::string &s) {
    printf("%s", s.c_str());
});
```

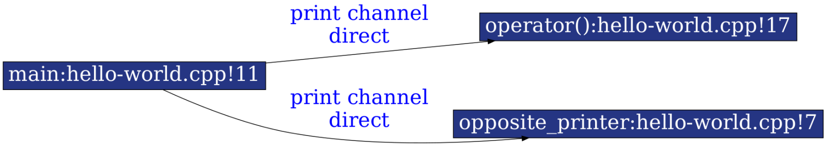
Producer:

```
//                                 signature           name
//                                    ↓                 ↓
auto print_channel = reg.lookup<void(std::string)>("print channel");
// deliver message now
print_channel("hello, world\n");
// deliver message in 1 cycle
//      ↓
sched(1, print_channel, "hello, world\n");
```

Example

```cpp
void opposite_printer(const std::string &s) {
    std::cout << (s == "hello" ? "world" : "hello") << '\n';
}

int main() {
    conduit::Registrar reg("reg");

    // producer
    auto print_channel = reg.lookup<void(std::string)>("print channel", "print_channel producer");

    // first consumer
    print_channel.hook([] (const std::string &s) {
        std::cout << s << '\n';
    });

    // second consumer
    print_channel.hook(opposite_printer);

    print_channel("hello");
}
```

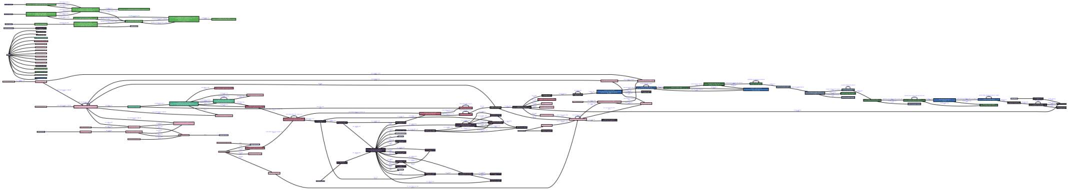
Reconstructing the decoupled call-graph



↓

```
print_channel producer -> reg.print channel(hello)
hello
world
```

Real example



Information recognized by the static analyzer

Idioms:

- Synchronous and asynchronous connections between components.
- Concurrent state collection (events may happen 0 to N times).
- Channel merge (wait for N different events before triggering).
- Comment processing to allow better semantic descriptions of execution elements.

State information:

- Non-const data members used in hook call-tree.

Benefits

- Provides programmers another level of abstraction with which to describe the problem.

    - This is now part of our "modelers contract": descriptive problem decomposition must be reflected through the static analysis (used to bridge the gap between software model and HW implementer).

- Reinforces event $\rightarrow$ state relationship.
- Helps identify concurrent data races.

Conclusion

Static analysis combined with programming convention allows reconstruction of data-flow across asynchronous boundaries.

Qualcomm

Thank you

Follow us on:

For more information, visit us at:

www.qualcomm.com & www.qualcomm.com/blog