ioa++

Justin R. Wilson

Table of Contents

1 Introduction

ioa++ is a general-purpose framework for developing asynchronous and concurrent programs based on the I/O automata model. Developers using ioa++ construct programs by defining and assembling event-based modules called I/O automata to form interacting constellations. As suggested by the name, ioa++ is implemented in C++.

1.1 The I/O Automata Model

The I/O automata model was developed by Nancy Lynch for asynchronous and concurrent systems and is described in Chapter 8 of *Distributed Algorithms*. I/O automata have been used to model and verify a number of real-world systems and protocols.

An I/O automaton consists of state variables and a set of atomic input, output, and internal actions. The set of actions in an automaton is known as its *signature*. Output and internal actions constitute the automaton's *local signature*. Local actions have a predicate over the state variables of the automaton (called a *precondition*) that indicates if the action can be executed. All actions have an *effect* which updates the state variables of the corresponding automaton. Input and output actions constitute the automaton's *external signature*.

I/O automata can be composed to form a new automaton by concatenating state variables and folding input actions into similarly named output actions. Whereas local actions can be *enabled* or *disabled* based on their precondition, input actions are executed whenever their associated output is executed. This property is known as being *input enabled*.

Execution proceeds by repeatedly selecting a local action and executing it if enabled. The model admits non-determinism by allowing the scheduler to pick actions in any order. Schedulers in I/O automata must be *fair* meaning they select (but do not necessarily execute) every action infinitely often.

1.2 Representing I/O Automata

I/O automata are encoded directly in the C++ programming language as classes that inherit from `ioa::automaton`. The state variables of the I/O automaton are naturally encoded as member variables of the class. The actions of the I/O automaton are encoded as a combination of member functions and member variables. A suite of templates and macros exist to simplify the definition of actions.

1.3 Dynamics

The I/O automata model assumes that systems consist of a static set of automata. Often, the size of the set is countably infinite meaning that it can be represented by an integer variable N. A model specified in this way is capable of capturing any real system since there is a countable number of participants in all real systems. Within an countably infinite set, a dynamic set of automata is simulated by associating a flag with each automaton indicating if it is active or inactive and defining appropriate "wake-up" and "sleep" actions. A direct implementation of this strategy does not work for real systems because an implementation must specify a concrete value for N. Computing with a fixed number of automata is either unduly prohibitive if resources exist for additional automata or unduly wasteful if the number of active automata is much less than N.

Consequently, we require the ability to dynamically create and destroy automata. An automaton that creates another automaton is called the *parent* while the created automaton is called the *child*. The automaton at the top of the hierarchy is called the *root*. Since we can dynamically create and destroy automata, we also require the ability to dynamically compose and decompose. *Binding* and *unbinding* refer to the act of dynamically composing and decomposing, respectively. For simplicity, binding and unbinding is limited to a single output action-input action pair. Explicit binding allows us to drop the requirement that the actions have the same name.

1.4 Run-time System

The ioa++ run-time system consists of a system automaton, a dispatcher, a scheduler, and a user-space library.

The *system automaton* contains the set of automata and bindings that exist in the system and actions for creating, binding, unbinding, and destroying. These actions are collectively called *system actions*. Each automaton is *composed* with the system automaton by inheriting from `ioa::automaton`.

The *dispatcher* enforces atomic execution according to the I/O automata model. Internal actions are executed by evaluating the precondition and then applying the effect if the precondition is true. The precondition and effect are computed in one atomic step. Output actions are executed similarly except that all bound input actions are evaluated atomically with the output action.

The *scheduler* selects the next action from a set of local actions. The set of local actions is updated by the automata in the system via `ioa::schedule` calls. Different schedulers can be used to realize different scheduling policies.

The user-space library is designed to help users write actions, schedule actions, and request system actions. Local actions consist of an object and three functions: a precondition, an effect, and a scheduling function. Input actions consists of an object and two functions: an effect and a scheduling function. The scheduling function is invoked after the effect of all actions and is intended as a place to schedule actions. The user-space library also contains classes that hide the complexities of creating and binding asynchronously.

1.5 Concurrency

Each local action involves a set of automata. For internal actions, this is just the automaton that contains the internal actions. For output actions, the set consists of the output action and the automata that contain the input actions bound to the output action. The semantics of I/O automata are such that two actions can be executed concurrently if their respective sets of involved automata are disjoint. An important consequence is that two actions belonging to the same automaton will never execute concurrently. True concurrent execution requires a scheduler capable of true concurrent execution, i.e., a multi-threaded scheduler.

1.6 Actions and Values

Recall that there are three types of actions: output actions, input actions, and internal actions. Output actions produce a signal or value, input actions consume a signal or value, and internal actions neither produce nor consume signals or values. Actions that produce or consume signals are said to be *unvalued* while signals that produce and consume values are said to be *valued*. External actions can only be bound together if they agree on the signal or type of value to be produced. Any input that consumes a signal can be bound to any output producing a signal. An input that consumes values of type T can only be bound to an output that produces a value of type T.

1.7 Parameters and Automatic Parameters

A common technique in the I/O automata model is to associate a parameter with an action. For example, actions that receive messages are often parameterized with communication endpoints. Parameters are distinct from values because they are constant under composition. All actions types, outputs, inputs, and internals, can be parameterized. Actions requiring a parameter are said to be *parameterized* while actions that don't require parameters are said to be *unparameterized*. Parameters must be used to identify parameterized actions. For example, the parameter for a parameterized output must specified when scheduling and binding. Similarly, a parameter for a parameterized input must specified when binding and a parameter for a parameterized internal must be specified when scheduling.

Parameters allow users to implement fan-in by associating a different parameter with each bind to an input. More generally, parameters can be used to implement a session by associating the same parameter with all bindings related to some automaton. Each automaton has a unique identifier called an *automaton identfier* or *aid*. Often, the parameter for a session is the aid of another automaton. To prevent errors and make sessions easier to implement, we introduce the concept of an automatic parameter. An *automatic parameter* or *auto parameter* is a parameter that represents the automaton on the opposite side of a binding. For example, an auto parameterized input bears the identifier of the output automaton to which it is bound. An auto parameterized output bears the identifier of the input automaton to which it is bound. According to the binding rules below, auto parameterized outputs can only be bound once.

1.8 Binding Rules

To enforce the semantics of I/O automata, certain attempts to bind will fail. The automaton requesting a binding is called the *owner*. A binding is a tuple (output automaton, output action, output parameter, input automaton, input action, input parameter, owner). Unparameterized actions have a null parameter. A binding will fail if any of the following conditions is true:

1. The owner does not exist.
2. The output automaton does not exist.
3. The input automaton does not exist.
4. The binding already exists. (This is only reported to the owner.)
5. The (input automaton, input action, input parameter) is already bound.
6. The (output automaton, output action, output parameter) is already bound to some input action in the input automaton.
7. The output automaton and input automaton are the same.

2 Examples

2.1 Compiling and Linking

The purpose of this tutorial is to introduce the necessary machinery for compiling programs with ioa++. The following program contains the null automaton—an automaton with no actions. The source can be found in '`tutorial/null_automaton.cpp`'.

```
#include <ioa/ioa.hpp>
#include <ioa/global_fifo_scheduler.hpp>

class null_automaton :
  public ioa::automaton
{ };

int main () {
  ioa::global_fifo_scheduler sched;
  ioa::run (sched, ioa::make_allocator<null_automaton> ());
  return 0;
}
```

Let's go through it section by section. The lines

```
#include <ioa/ioa.hpp>
#include <ioa/global_fifo_scheduler.hpp>
```

include all of the headers necessary for writing I/O automata and the header needed to declare a global FIFO scheduler. The lines

```
class null_automaton :
  public ioa::automaton
{ };
```

declare a new automaton type called `null_automaton`. All automata must inherit from `ioa::automaton` (see [ioa::automaton], page 21). Also note that all ioa++ types and functions are in the `ioa` namespace. The main function

```
int main () {
  ioa::global_fifo_scheduler sched;
  ioa::run (sched, ioa::make_allocator<null_automaton> ());
  return 0;
}
```

declares a new scheduler `sched` of type `ioa::global_fifo_scheduler` (see [ioa::global_fifo_scheduler], page 21) and starts the scheduler with a new root automaton of type `null_automaton`. The `ioa::run` function takes two arguments: a scheduler and an allocator (see [ioa::run], page 21). An allocator is an object that can later be invoked to produce a dynamically allocated object (see [ioa::make_allocator], page 21). In this case, the allocator returns a dynamically created instance of `null_automaton`.

Assuming that a copy of '`null_automaton.cpp`' exists in the current directory and that `g++` is your C++ compiler, one can compile and run the null automaton with

```
$ g++ null_automaton.cpp -o null_automaton -lioa -lpthread
$ ./null_automaton
```

Notice that we needed to link against the I/O automata library (‘`-lioa`’) and pthreads library (‘`-lpthread`’). Some environments, e.g., Mac OS X, include pthreads in the standard C library. If you have such an environment, omit the ‘`-lpthread`’ part of the command.

2.2 Internal Actions

In this tutorial we develop an automaton that counts to ten using an internal action. The source is given below and can be found in ‘`tutorial/count_to_ten_automaton.cpp`’.

```
#include <ioa/ioa.hpp>
#include <ioa/global_fifo_scheduler.hpp>

// For std::cout.
#include <iostream>

class count_to_ten_automaton :
  public ioa::automaton
{
private:
  int m_count;

public:
  count_to_ten_automaton () :
    m_count (1) {
    increment_schedule ();
  }

private:
  bool increment_precondition () const {
    return m_count <= 10;
  }

  void increment_effect () {
    std::cout << m_count << std::endl;
    ++m_count;
  }

  void increment_schedule () const {
    if (increment_precondition ()) {
      ioa::schedule (&count_to_ten_automaton::increment);
    }
  }

  UP_INTERNAL (count_to_ten_automaton, increment);
};
```

```
int main () {
  ioa::global_fifo_scheduler sched;
  ioa::run (sched, ioa::make_allocator<count_to_ten_automaton> ());
  return 0;
}
```

The automata in this tutorial are listed in a way that attempts to mimic the style in *Distributed Algorithms*. In general, an automaton will have the following structure:

- Type definitions — Declare types that are used internally by the automaton and types that are used by external actions.
- State declarations — Declare the state variables of the automaton.
- Constructors/Destructors — Declare/define constructors to initialize the state variables and destructors to perform any required clean-up.
- Private member functions — Declare/define useful functions.
- Actions — Declare/define the actions of the automaton. Local actions consist of a precondition, an effect, and a scheduling function. Input actions consist of an effect and a scheduling function.

Let's examine the automaton section by section. The state of the automaton is declared with

```
private:
  int m_count;
```

In this case, the state of the automaton consists of a single integer `m_count`. State variables should always be declared `private`.

The constructor initializes the count to 1 and calls the `increment_schedule` member function which is described below:

```
public:
  count_to_ten_automaton () :
    m_count (1) {
    increment_schedule ();
  }
```

The next section defines the precondition, effect, and scheduling function for an unparameterized internal action named `increment`:

```
  bool increment_precondition () const {
    return m_count <= 10;
  }

  void increment_effect () {
    std::cout << m_count << std::endl;
    ++m_count;
  }

  void increment_schedule () const {
    if (increment_precondition ()) {
      ioa::schedule (&count_to_ten_automaton::increment);
    }
```

```
    }
```

Following the style in *Distributed Algorithms*, internal actions are divided into a *precondition* and *effect*. The precondition returns a `bool` indicating if the action can be executed. In this example, the precondition returns true so long as the count is less than or equal to ten. The pattern for declaring the precondition for an unparameterized internal actions is `bool` *`action-name`*`_precondition () const`. Note that preconditions have the `const` modifier as they should not change the state of the automaton. The effect changes the state of the automaton. In this example, the effect prints the current value of the count and increments the count. The pattern for declaring the effect of an unparameterized internal action is `void` *`action-name`*`_effect ()`. The scheduling function is called after the effect and it used to tell the scheduler about actions that should be selected. SCHEDULE see [ioa::schedule], page 21 In this example, the `increment` action is scheduled if its precondition is true. The pattern for declaring a scheduling fuctions is `void` *`action-name`*`_schedule () const`. Preconditions, effects, and scheduling functions should always be declared `private`.

The final part of declaring/defining an unparameterized internal action is to declare a member variable representing the action that dispatches to the precondition, effect, and scheduling function and also contains appropriate typedefs for the scheduler. This is tedious so a set of a macros is defined to simplify declaring the members. The code

```
    UP_INTERNAL (count_to_ten_automaton, increment);
```

uses the `UP_INTERNAL` macro to declare an action member variable `increment`. The `UP_INTERNAL` macro and similar macros rely on the `*_precondition`, `*_action`, and `*_schedule` naming convention described earlier. The macro arranges for the `*_schedule` member function to be called after each action effect. Internal actions, i.e., the scope where `UP_INTERNAL` appears, should be `private`. External actions can either be `private`, `protected`, or `public` depending on their intended use.

To summarize, consider the uses of the automaton name and action names. To declare an automaton:

```
class automaton-name : public ioa::automaton ...
```

To declare a precondition for an unparameterized internal action:

```
bool action-name_precondition () const
```

To declare a effect for an unparameterized internal action:

```
void action-name_effect ()
```

To declare a scheduling function:

```
void action-name_schedule () const
```

To declare an unparameterized internal action:

```
UP_INTERNAL (automaton-name, action-name)
```

To schedule an unparameterized local action:

```
ioa::schedule (&automaton-name::action-name)
```

Programming Tip: Forgetting to schedule is common source of problems when programming with ioa++. Remember to call `ioa::schedule` for all enabled local actions in the constructor and scheduling functions (or effects). As an exercise, experiment with commenting out the call to `increment_schedule` in the constructor and the call to `ioa::schedule` in `increment_schedule`.

HALTING

2.3 Creating Automata

Decomposition is a powerful technique for managing complexity—especially in concurrent and distributed systems. Instead of solving the problem with one large automaton, we can decompose the problem into a number of simpler automata and bind their actions together. Additionally, decomposition also allows us to find generic components that can be reused for many problems.

In this tutorial we develop an automaton that creates two `count_to_ten_automaton`s. Binding actions will be covered later when we cover input and output actions. The source is given below and can be found in '`tutorial/two_counters.cpp`'.

```
/*
   Copyright 2011 Justin R. Wilson

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
*/

#include <ioa/ioa.hpp>
#include <ioa/global_fifo_scheduler.hpp>

#include <iostream>

class count_to_ten_automaton :
  public ioa::automaton
{
private:
  int m_count;

public:
  count_to_ten_automaton () :
    m_count (1)
  {
    increment_schedule ();
  }

private:
  bool increment_precondition () const {
    return m_count <= 10;
```

```
    }

    void increment_effect () {
      std::cout <<
        "automaton:  " << ioa::get_aid () <<
        " count:  " << m_count << std::endl;
      ++m_count;
    }

    void increment_schedule () const {
      if (increment_precondition ()) {
        ioa::schedule (&count_to_ten_automaton::increment);
      }
    }

    UP_INTERNAL (count_to_ten_automaton, increment);
  };

  class two_counter_automaton :
    public ioa::automaton
  {
  public:
    two_counter_automaton () {
      ioa::make_automaton_manager (this,
          ioa::make_allocator<count_to_ten_automaton> ());
      ioa::make_automaton_manager (this,
          ioa::make_allocator<count_to_ten_automaton> ());
    }

  };

  int main () {
    ioa::global_fifo_scheduler sched;
    ioa::run (sched, ioa::make_allocator<two_counter_automaton> ());
    return 0;
  }
```

This example contains two automata: `count_to_ten_automaton` and `two_counter_automaton`. Of primary interest is the constructor of the `two_counter_automaton`.

```
    two_counter_automaton () {
      ioa::make_automaton_manager (this,
          ioa::make_generator<count_to_ten_automaton> ());
      ioa::make_automaton_manager (this,
          ioa::make_generator<count_to_ten_automaton> ());
    }
```

The function `ioa::make_automaton_manager` creates a dynamically allocated `ioa::automaton_manager`. An `ioa::automaton_manager` uses an `ioa::automaton` object and generator to *asynchronously* create a new automaton. *Automaton creation and destruction are asynchronous in ioa++.* A pointer to an `ioa::automaton` object is specified by the first argument to the `ioa::make_automaton_manager` function. This should always be the `this` pointer of the automaton that is creating a new automaton. The second argument is a generator that returns an instance of the automaton to be created. In this example, each generator returns a `count_to_ten` automaton.

If automaton A creates automaton B then A is the *parent* of B and B is the *child* of A. Thus, automata in ioa++ form a tree with the automaton generated by `ioa::run` being the root. When an automaton is destroyed, so are all of its children.

Something that might concern you is that fact that the automaton managers are dynamically allocated but we neither save their address nor `delete` them in a destructor. Upon construction, the `ioa::automaton` object takes ownership of the automaton manager. A parent automaton can request that one of its children be destroyed using its `destroy` method and can detect child automata that have been destroyed by observing the automaton manager. Note that a child automaton might voluntarily destroy itself, i.e., self destruct, when it has no more work to do. A parent automaton should forget the automaton manager of any child that has been destroyed.

To distinguish the output of the two `count_to_ten_automatons`, we use the `ioa::get_aid` function.

```
std::cout <<
  "automaton: " << ioa::get_aid () <<
  " count: " << m_count << std::endl;
```

Associated with each instance of an automaton is an *automaton identifier (aid)* of type `aid_t` that can be retrieved with `ioa::get_aid`.

2.4 External Actions

In this tutorial, we introduce input and output actions and the concept of binding using a simple producer-consumer problem. We split the `count_to_ten_automaton` in previous tutorials into an automaton that produces numbers and another automaton that prints numbers. The source is given below and can be found in '`tutorial/producer_consumer.cpp`'.

```
/*
   Copyright 2011 Justin R. Wilson

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
*/
```

```
#include <ioa/ioa.hpp>
#include <ioa/global_fifo_scheduler.hpp>

#include <iostream>

class producer_automaton :
  public ioa::automaton
{
private:
  int m_count;

public:
  producer_automaton () :
    m_count (1) {
    produce_schedule ();
  }

private:
  bool produce_precondition () const {
    return m_count <= 10;
  }

  int produce_effect () {
    int retval = m_count++;
    std::cout << "producing " << retval << std::endl;
    return retval;
  }

  void produce_schedule () const {
    if (produce_precondition ()) {
      ioa::schedule (&producer_automaton::produce);
    }
  }

public:
  V_UP_OUTPUT (producer_automaton, produce, int);
};

class consumer_automaton :
  public ioa::automaton
{
private:
  void consume_effect (const int& val) {
    std::cout << "consuming " << val << std::endl;
  }
```

```
    void consume_schedule () const { }

  public:
    V_UP_INPUT (consumer_automaton, consume, int);
  };

  class producer_consumer_automaton :
    public ioa::automaton
  {
  public:
    producer_consumer_automaton () {
      ioa::automaton_manager<producer_automaton>* producer =
        ioa::make_automaton_manager (this,
            ioa::make_allocator<producer_automaton> ());

      ioa::automaton_manager<consumer_automaton>* consumer =
        ioa::make_automaton_manager (this,
            ioa::make_allocator<consumer_automaton> ());

      ioa::make_binding_manager (this,
                                 producer, &producer_automaton::produce,
                                 consumer, &consumer_automaton::consume);
    }

  };

  int main () {
    ioa::global_fifo_scheduler sched;
    ioa::run (sched, ioa::make_allocator<producer_consumer_automaton> ());
    return 0;
  }
```

The first part of the `producer_automaton` resembles the `count_to_ten_automaton` in previous tutorials where we have renamed the `increment` action to `produce`. Of interest in this tutorial is `produce_effect` and the declaration of the `produce` output action:

```
    int produce_effect () {
      int retval = m_count++;
      std::cout << "producing " << retval << std::endl;
      return retval;
    }

  public:
    V_UP_OUTPUT (producer_automaton, produce, int);
```

The `produce_effect` increments the counter and returns the old value of the counter. The macro `V_UP_OUTPUT` declares a valued unparameterized output action named `produce` that produces a value of type `int`. The `V_UP_OUTPUT` macro relies on the same naming conventions as the `UP_INTERNAL` action seen in preceding tutorials. As an exercise, make the type given to `V_UP_OUTPUT` different from the type returned by `produce_effect`, e.g., change `int` to `float`, and recompile. Note that `V_UP_OUTPUT` appears in a `public` section. This is necessary because we want to allow other automata to bind to this action.

The `consumer_automaton` is quite simple and only contains a single valued unparameterized input action:

```
    void consume_effect (const int& val) {
      std::cout << "consuming " << val << std::endl;
    }

    void consume_schedule () const { }

  public:
    V_UP_INPUT (consumer_automaton, consume, int);
```

Recall that I/O automata are input enabled so there is no `consume_precondition`. The macro `V_UP_INPUT` declares a valued unparameterized input action named `consume` that takes a value of type `int`. The `V_UP_INPUT` macro relies on the same naming conventions as the other macros. The effect used by `V_UP_INPUT` must take a single argument declared as a constant reference. Compare this with `consume_effect`. Again, note that `V_UP_INPUT` appears in a `public` section so we can bind to it.

The `consume_schedule` function is required even though it is empty. As an exercise, comment out the `consume_schedule` function and recompile to become familiar with the compilation error caused by omitting it.

The constructor of the `producer_consumer_automaton` creates a `producer_automaton` and a `consumer_automaton` and then binds the `produce` and `consume` actions of the respective automatons together:

```
    producer_consumer_automaton () {
      ioa::automaton_manager<producer_automaton>* producer =
        ioa::make_automaton_manager (this,
            ioa::make_generator<producer_automaton> ());

      ioa::automaton_manager<consumer_automaton>* consumer =
        ioa::make_automaton_manager (this,
            ioa::make_generator<consumer_automaton> ());

      ioa::make_binding_manager (this,
                                 producer, &producer_automaton::produce,
                                 consumer, &consumer_automaton::consume);
    }
```

The child automata are created in the same way as in the preceding tutorials only we save the pointer to the new manager so we can pass it to the `ioa::make_binding_manager` function. This version of `ioa::make_binding_manager` takes a pointer to an `ioa::automaton` (see Section 2.3 [Creating Automata], page 9), a pointer to an `ioa::automaton_handle_interface` for the output automaton, a pointer to a member for the output action, a pointer to an `ioa::automaton_handle_interface` for the input automaton, and a pointer to a member for the input action. The `ioa::automaton_handle_interface` is unimportant save to say that `ioa::automaton_manager` implements `ioa::automaton_handle_interface`. The `ioa;::make_binding_manager` function creates a new `ioa::binding_manager` that binds the output and input actions once the output and input automata have been created. Note that like automata creation, binding is asynchronous in ioa++.

2.4.1 Binding Rules

There are a number of rules that must be observed when binding. The first set of rules are checked at compile time.

1. One Output, One Input — The first automaton/action pair must be an output and the second automaton/action pair must be an input.
2. Access — The output and input must be accessible from the scope where `ioa::make_binding_helper` is invoked. For example, if automaton C is binding an output action in automaton O to an input action in automaton I, then both of the actions must be declared `public`. As another example, if automaton C is binding one of its own output actions to an input in automaton I, then C can (and probably should) declare the output to be `private`.
3. Value Status Agreement — External actions need not produce/consume values. External actions that don't produce/consume values are called *unvalued* while external action that do produce/consume values are called *valued*. When binding, both the output and the input action must have the same value status.
4. Type Agreement — Valued external actions must agree on the same type. For example, the `produce` action and `consume` action agree that the value being produced/consumed is an `int`.

The second set of rules are checked at run time.

1. An input action can only be bound to one output action.
2. An output action cannot be bound to two different inputs residing in the same automaton.
3. An output action cannot be bound to an input action in the same automaton.

These rules are necessary to adhere to the I/O automata model. Recall that automata are composed by matching the names of output and input actions. One could imagine a function that rewrites the names of all actions before composing. These rules say that such a function exists.

2.4.2 Binding Dynamics

When I execute `producer_consumer`, I get the following output:

```
$ ./producer_consumer
producing 1
producing 2
producing 3
producing 4
producing 5
producing 6
producing 7
```

```
consuming 7
producing 8
consuming 8
producing 9
consuming 9
producing 10
consuming 10
```

Recall that automata creation and binding is asynchronous. In this example, the producer was allowed to `produce` six times before `produce` was bound to `consume`. The `produce` action is a *lossy output* or an output whose values might be lost because no input is bound to receive them. We address this topic in the next tutorial.

2.5 Fan-out and Binding Count

In this tutorial, we bind an output action to multiple input actions (fan-out) and prevent lost outputs by counting the number of bindings associated with an action. The source is given below and can be found in '`tutorial/producer_consumer2.cpp`'.

```
/*
   Copyright 2011 Justin R. Wilson

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
*/

#include <ioa/ioa.hpp>
#include <ioa/global_fifo_scheduler.hpp>

#include <iostream>

class producer_automaton :
  public ioa::automaton
{
private:
  int m_count;

public:
  producer_automaton () :
    m_count (1) { }
```

```
private:
  bool produce_precondition () const {
    return m_count <= 10 &&
      ioa::binding_count (&producer_automaton::produce) == 2;
  }

  int produce_effect () {
    int retval = m_count++;
    std::cout << "producing " << retval << std::endl;
    return retval;
  }

  void produce_schedule () const {
    if (produce_precondition ()) {
      ioa::schedule (&producer_automaton::produce);
    }
  }

public:
  V_UP_OUTPUT (producer_automaton, produce, int);
};

class consumer_automaton :
  public ioa::automaton
{
private:
  void consume_effect (const int& val) {
    std::cout << ioa::get_aid () << " consuming " << val << std::endl;
  }

  void consume_schedule () const { }

public:
  V_UP_INPUT (consumer_automaton, consume, int);
};

class producer_consumer_automaton :
  public ioa::automaton
{
public:
  producer_consumer_automaton () {
    ioa::automaton_manager<producer_automaton>* producer =
      ioa::make_automaton_manager (this,
          ioa::make_allocator<producer_automaton> ());

    ioa::automaton_manager<consumer_automaton>* consumer1 =
```

```
      ioa::make_automaton_manager (this,
          ioa::make_allocator<consumer_automaton> ());

    ioa::automaton_manager<consumer_automaton>* consumer2 =
      ioa::make_automaton_manager (this,
          ioa::make_allocator<consumer_automaton> ());

    ioa::make_binding_manager (this,
                               producer, &producer_automaton::produce,
                               consumer1, &consumer_automaton::consume);

    ioa::make_binding_manager (this,
                               producer, &producer_automaton::produce,
                               consumer2, &consumer_automaton::consume);
  }

};

int main () {
  ioa::global_fifo_scheduler sched;
  ioa::run (sched, ioa::make_allocator<producer_consumer_automaton> ());
  return 0;
}
```

The constructor of the producer_consumer_automaton creates a producer_automaton and two consumer_automatons and then binds the produce and consume actions of the respective automatons together:

```
  producer_consumer_automaton () {
    ioa::automaton_manager<producer_automaton>* producer =
      ioa::make_automaton_manager (this,
          ioa::make_generator<producer_automaton> ());

    ioa::automaton_manager<consumer_automaton>* consumer1 =
      ioa::make_automaton_manager (this,
          ioa::make_generator<consumer_automaton> ());

    ioa::automaton_manager<consumer_automaton>* consumer2 =
      ioa::make_automaton_manager (this,
          ioa::make_generator<consumer_automaton> ());

    ioa::make_binding_manager (this,
                               producer, &producer_automaton::produce,
                               consumer1, &consumer_automaton::consume);

    ioa::make_binding_manager (this,
                               producer, &producer_automaton::produce,
                               consumer2, &consumer_automaton::consume);
```

```
}
```

We changed the `consume_effect` of the `consumer_automaton` to print out the aid to distinguish between the two consumers:

```
void consume_effect (const int& val) {
  std::cout << ioa::get_aid () << " consuming " << val << std::endl;
}
```

We also removed the call to `produce_schedule` in the `producer_automaton` constructor:

```
producer_automaton () :
  m_count (1) { }
```

More on this later.

The most important change is the addition of an `ioa::binding_count` to the `produce_precondition` of the `producer` automaton:

```
bool produce_precondition () const {
  return m_count <= 10 &&
    ioa::binding_count (&producer_automaton::produce) == 2;
}
```

The `ioa::binding_count` returns the number of actions to which the given action is bound. The binding count of an internal action will always be 0. The binding count of an output action is a non-negative integer. The binding count of an input action is either 0 or 1. In this example, the `produce_effect` is not executed until the two consumers have been bound.

You might be wondering, "If we don't schedule the `produce` action in the constructor, how does it get executed?" The answer is that scheduler automatically schedules output actions when they are bound. Thus, whenever the second automaton binds to `produce`, `produce` is scheduled. Since the binding count is now 2, the `produce_precondition` becomes true, and `produce_effect` and `produce_schedule` are evaluated. Scheduling output actions in this way takes advantage of the concept of a fair scheduler in the I/O automata model and seems to be a graceful way of handling the common case where an output must be bound before it is executed.

When I execute `producer_consumer2`, I get the following output:

```
$ ./producer_consumer2
producing 1
3 consuming 1
4 consuming 1
producing 2
3 consuming 2
4 consuming 2
producing 3
3 consuming 3
4 consuming 3
producing 4
3 consuming 4
4 consuming 4
producing 5
```

```
3 consuming 5
4 consuming 5
producing 6
3 consuming 6
4 consuming 6
producing 7
3 consuming 7
4 consuming 7
producing 8
3 consuming 8
4 consuming 8
producing 9
3 consuming 9
4 consuming 9
producing 10
3 consuming 10
4 consuming 10
```

which shows that both consumers receive the values produced by the producer.

3 Reference

`ioa::automaton` [Class]

The base class for all automata that implements the system action interface which allows an automaton to request system actions and receive system action results. All automata must publicly inherit from `ioa::automaton`. From '`<ioa/automaton.hpp>`'.

`ioa::global_fifo_scheduler` [Class]

A single-threaded scheduler that implements the first-in/first-out (FIFO) policy. From '`<ioa/global_fifo_scheduler.hpp>`'.

`template <class T> void ioa::run (scheduler_interface& sched, std::auto_ptr<typed_allocator_interface<T> > allocator)` [Function]

Starts the scheduler *sched* with the root automaton produced by *allocator*. From '`<ioa/scheduler.hpp>`'.

`template <class T, A0, ...> std::auto_ptr<typed_allocator_interface<T, A0, ...> > make_allocator (A0 a0, ...)` [Function]

A set of helper functions for allocating allocators. This set of functions takes advantage of the automatic type deduction rules of C++ to avoid repeating type information. From '`<ioa/allocator.hpp>`'.

- ioa::schedule
- ioa::make_automaton_manager
- ioa::automaton_manager
- ioa::get_aid
- ioa::aid_t
- ioa::make_binding_manager
- ioa::automaton_handle_interface
- ioa::binding_manager
- UV_UP_INPUT
- UV_P_INPUT
- UV_AP_INPUT
- V_UP_INPUT
- V_P_INPUT
- V_AP_INPUT
- UV_UP_OUTPUT
- UV_P_OUTPUT
- UV_AP_OUTPUT
- V_UP_OUTPUT
- V_P_OUTPUT
- V_AP_OUTPUT
- UP_INTERNAL

- P_INTERNAL
- SYSTEM_INPUT
- SYSTEM_OUTPUT

Type Index

Function Index

I

M

Copying

is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the

Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.