Compile-time Algorithms On Overload Sets

Alexander Nasonov
alnsn@yandex.ru

ACCU 2005, Oxford, UK

What is overloading? (formal)

13 Overloading [over]

1 When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. [...]

2 When an overloaded function name is used in a call, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called overload resolution [...]

What is overloading? (mantra)

a + b ;

What is overloading after all?

- Complex but supposed to be intuitive
  - *Hint: watch mantra*
- At call time
- Best function selection by static types of arguments
  - *Sometimes called static dispatch*
- Often used
- Often misused
  - *Mixture with implicit conversions is highly explosive*
- *Exotic sizeof trick*
  - *Most complex trick is in boost::is_base_and_derived*

Refer to Overload 66, Overload Resolution – Selecting the Function by Mikael Kilpeläinen

How it can be extended?

- Disable/enable functions selectively
- Analyze signatures at compile-time
  - *Concept check of arguments and return types*
- Generate code from an overload set
  - *Multimethods*
  - *Finite State Machine*

Multimethods

```
class Shape { /* ... */ };
class Circle    : public Shape { /* ... */ };
class Rectangle : public Shape { /* ... */ };

bool has_intersection(Circle&,    Circle&   );
bool has_intersection(Circle&,    Rectangle&);
bool has_intersection(Rectangle&, Circle&   );
bool has_intersection(Rectangle&, Rectangle&);
// Non-const references only to save space
```

Multimethods (continued)

- Each function expects that **static** type of each argument is the same as its **dynamic** type
- They cover functionality of all possible combinations of concrete shapes
- There is no accept-any-shapes function:

```
bool has_intersection(Shape&, Shape&);
```

- There is no language support for generating it
- As a result, extra efforts to generate multiple dispatch code

What we need?

A tool that can analyze signatures of overloaded functions and generate dispatch code

How to get signature by function name?

```
template<class R, class T>
void get_signature(R(*)(T))
{
    typedef R(signature)(T);
}

int foo(char);
// char foo(int);

int main()
{
    get_signature(&foo);
}
```

Problems in red, resolved issues in blue:

- Signature is available only inside get_signature function
- Function foo is not overloaded
- Ambiguity error if foo was overloaded

How to get signature by function name?

```
#include <boost/mpl/identity.hpp>

template<class R, class T>
boost::mpl::identity<R(T)>
get_signature(R(*)(T));

int foo(char);
// char foo(int);

int main()
{
    // if typeof is available
    typedef typeof(
        get_signature(&foo)
      ) signature_id;

    typedef signature_id::type
        signature;
}
```

Problems in red, resolved issues in blue:

- Signature is available only inside get_signature function
- Function foo is not overloaded
- Ambiguity error if foo was overloaded

How to get signature by function name?

```
#include <boost/mpl/identity.hpp>

template<long N> struct id {};

template<long N, class R, class T>
boost::mpl::identity<R(T)>
get_signature(R(*)(id<N>, T));

int  foo(id<1>, char);
char foo(id<2>, int);

int main()
{
    typedef typeof(
        get_signature<1>(&foo)
      ) signature_id;

    typedef signature_id::type
        signature;
}
```

Problems in red, resolved issues in blue:

- Signature is available only inside get_signature function
- Function foo is not overloaded
- Ambiguity error if foo was overloaded
- New overload set differs significantly from original set

Back to original set

```
struct enable_all {};

template<long N> struct id
{
    // id<N> is constructible
    // from enable_all for any N
    id(enable_all) {}
};

int  foo(id<1>, char);
char foo(id<2>, int);

int main()
{
    foo(enable_all(), '1');
    foo(enable_all(),  1 );
}
```

Problems in red, resolved issues in blue:

- Signature is available only inside get_signature function
- Function foo is not overloaded
- Ambiguity error if foo was overloaded
- New overload set differs significantly from original set

Summary of technique

- First argument *id*<*N*> is specially crafted to identify a function
- All functions are enumerated starting from 1
- Get function signature by *id*<*N*>
- At call time, enable_all is passed instead of *id*<*N*> to mimic original overloading rules

Improvements

- Use type instead of function name
- Use const-qualified call-operators to represent an overload set
- Avoid typeof where possible
- Don't rely on function id
- Support MPL concepts
  - *id<N> is* ***IntegralConstant***
  - *overload set is* ***Associative Sequence***

Example

```
struct has_intersection
{
    bool operator()(id<1>, Circle&,    Circle&   ) const;
    bool operator()(id<2>, Circle&,    Rectangle&) const;
    bool operator()(id<3>, Rectangle&, Circle&   ) const;
    bool operator()(id<4>, Rectangle&, Rectangle&) const;
};

int main()
{
    Circle c;
    Rectangle r;
    has_intersection()(enable_all(), r, c);
}
```

overloads::set sequence

- It is MPL sequence that allow viewing an overload set as a sequence of function types
- MPL *Bidirectional Sequence* interface
  - *begin/end, size, empty, front/back*
  - *find/find_if, contains, equal, max_element etc*
- MPL *Associative Sequence* concept
  - *has_key, order, etc*
- MPL *Bidirectional Iterator* interface
  - *next/prior, deref*

(Metafunctions in red require typeof)

Three forms of overloads::set

```
// Natural overload set
typedef overloads::set<has_intersection> natural_set;

// Only overloads with ids in range [1, 5)
typedef overloads::set<
    has_intersection, mpl::range_c<int,1,5>
  > range_set;

// Only overloads with selected ids
typedef overloads::set<
    has_intersection
  , mpl::set< id<1>, id<2> >
  > selected_ids_set;
```

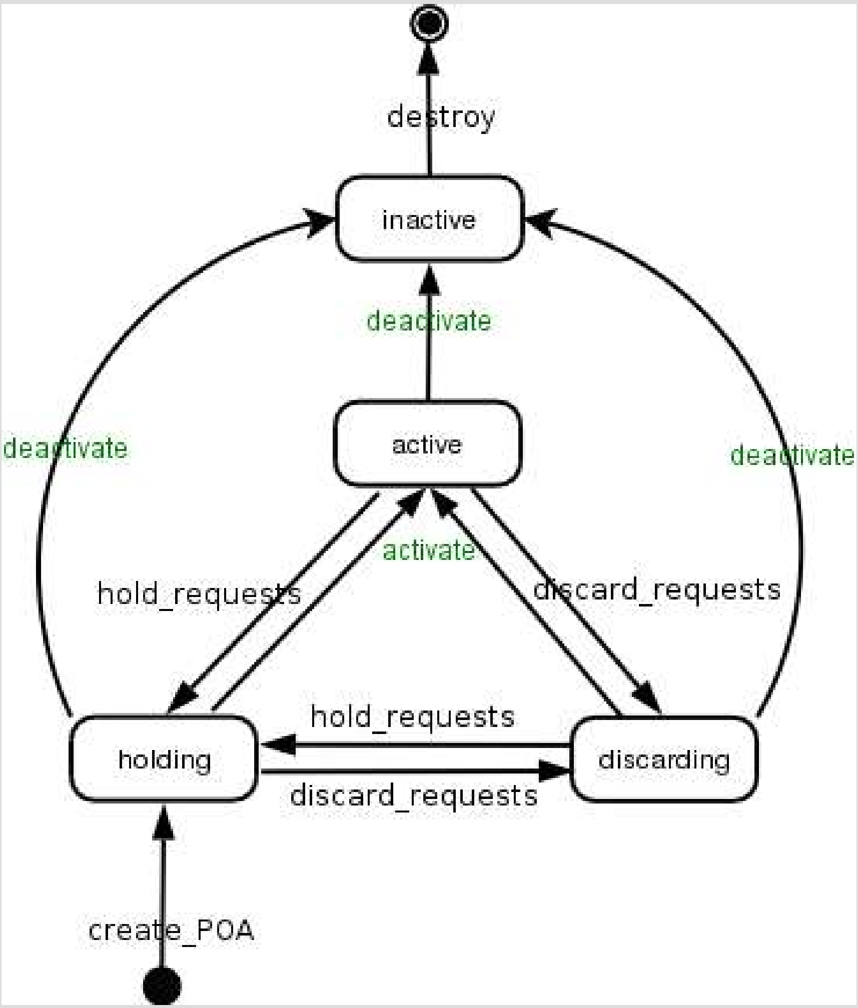
Associative Sequence interface

```
// mpl::has_key
BOOST_MPL_ASSERT((
    mpl::has_key<
        overloads::set<has_intersection>
      , bool(Circle&,Circle&)
      >
  ));

// mpl::order
typedef mpl::order<
    overloads::set<has_intersection>
  , bool(Circle&,Circle&)
  >::type order1;
BOOST_MPL_ASSERT(( mpl::equal_to<id<1>, order1> ));
```

FSM

Finite State Machine

POAManager life cycle (CORBA)



- Requirements are shown on FSM diagram
- Often implemented by hand, without using any FSM library

Classical approach

- Hand-crafted code to emulate state machine behavior
- State is a value of special member or combination of values
- All members are accessible even though some of them are valid only in specific state

As a result,

- Hard to maintain invariants
- Ensuring appropriate exception safety may become a serious problem
- Less readable or even messy code

FSM model

- State definitions (including initial states), event definitions and overload set of transitions fully define FSM
- Event is a type and may have data
- State is a type and may have data
- Transition is a function that accepts current state and event and returns new state:

```
S new_state = transition(current_state, event);
```

- Return type of transition function is always state
- Formal argument 2 of transition function has always same type as actual argument

Events aren't polymorphic in any sense of word polymorphism: neither static nor dynamic

FSM model (continued)

- Formal argument 1 of transition function doesn't necessarily have same type as actual argument, derived-to-base conversion may be applied
  - *To group transitions from several states with common base into one transition*
  - *To keep data common to several states in one place*
- Transition is transactional

```
S new_state = transition(current_state, event);
```

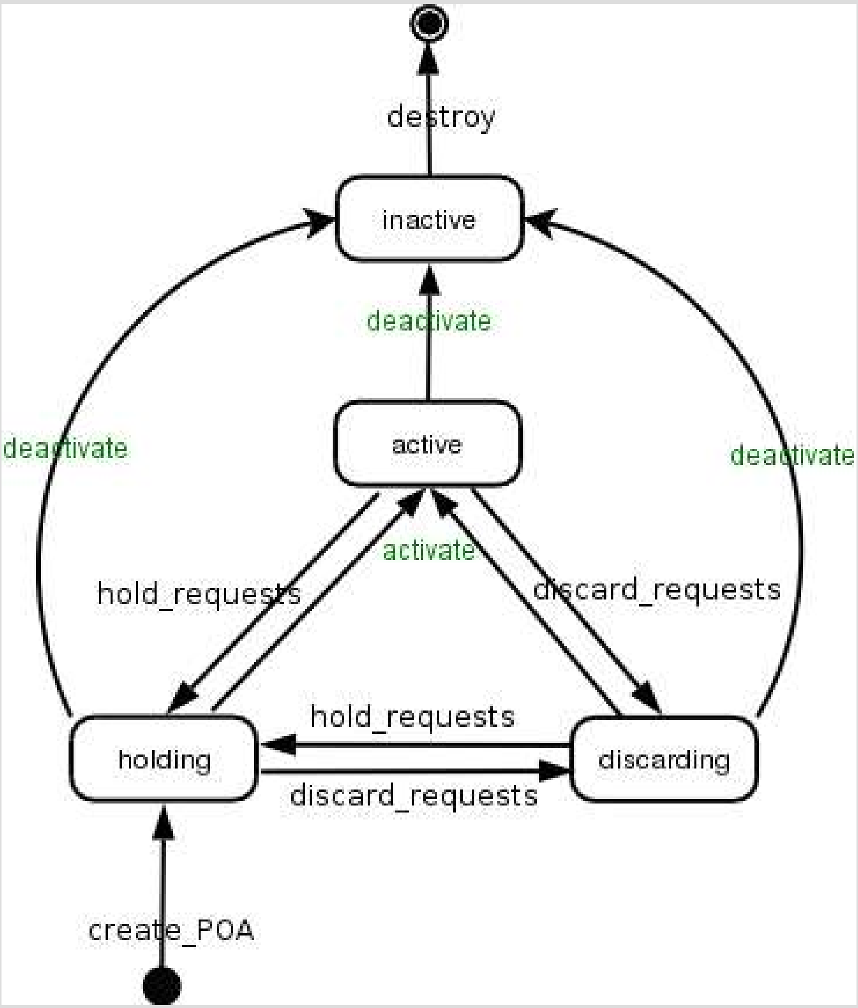
  - *New state is constructed*
  - *Current state is destroyed (can't fail)*
  - *State is changed to new state (can't fail)*
- If there's no transition, nothing happens

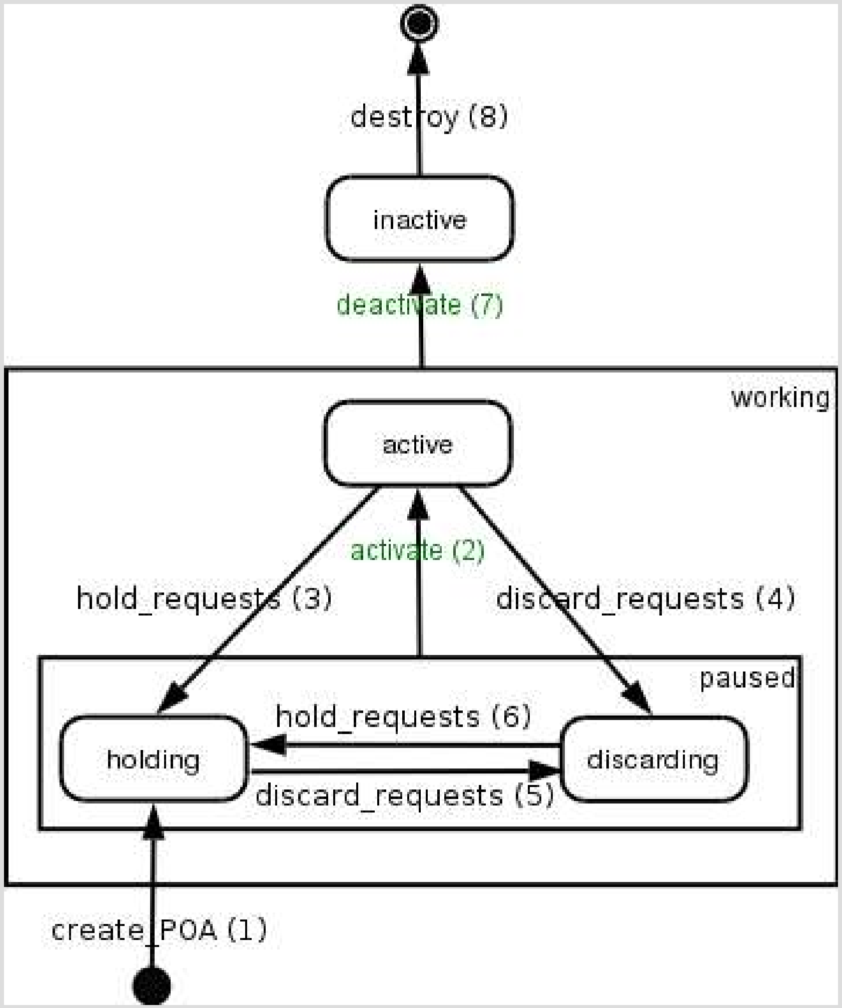
FSM model (continued)

- O(1) performance

  Transition is an access to transition matrix element followed by transition call
- Exception neutral
  - Strong exception safety
  - *Compiles even if exceptions are disabled*
- No RTTI required

States grouping





States

```
// State bases:
struct working_ { /* ... */ };
struct paused_ : working_ { /* ... */ };

// States:
struct start_     { /* ... */ };
struct finish_    { /* ... */ };
struct inactive_ { /* ... */ };
struct active_       : working_ { /* ... */ };
struct holding_      : paused_  { /* ... */ };
struct discarding_ : paused_  { /* ... */ };
```

Events

```
// Events:
struct create_POA_ { /* ... */ };
struct activate_ { /* ... */ };
struct deactivate_ { /* ... */ };
struct discard_requests_ { /* ... */ };
struct hold_requests_ { /* ... */ };
struct destroy_ { /* ... */ };
```

Transitions

```
struct transitions
{
    holding_    operator()(id<1>, start_,       create_POA_        ) const;
    active_     operator()(id<2>, paused_,      activate_          ) const;
    holding_    operator()(id<3>, active_,      hold_requests_     ) const;
    discarding_ operator()(id<4>, active_,      discard_requests_) const;
    discarding_ operator()(id<5>, holding_,     discard_requests_) const;
    holding_    operator()(id<6>, discarding_, hold_requests_     ) const;
    inactive_   operator()(id<7>, working_,     deactivate_        ) const;
    finish_     operator()(id<8>, inactive_,    destroy_           ) const;
};
```

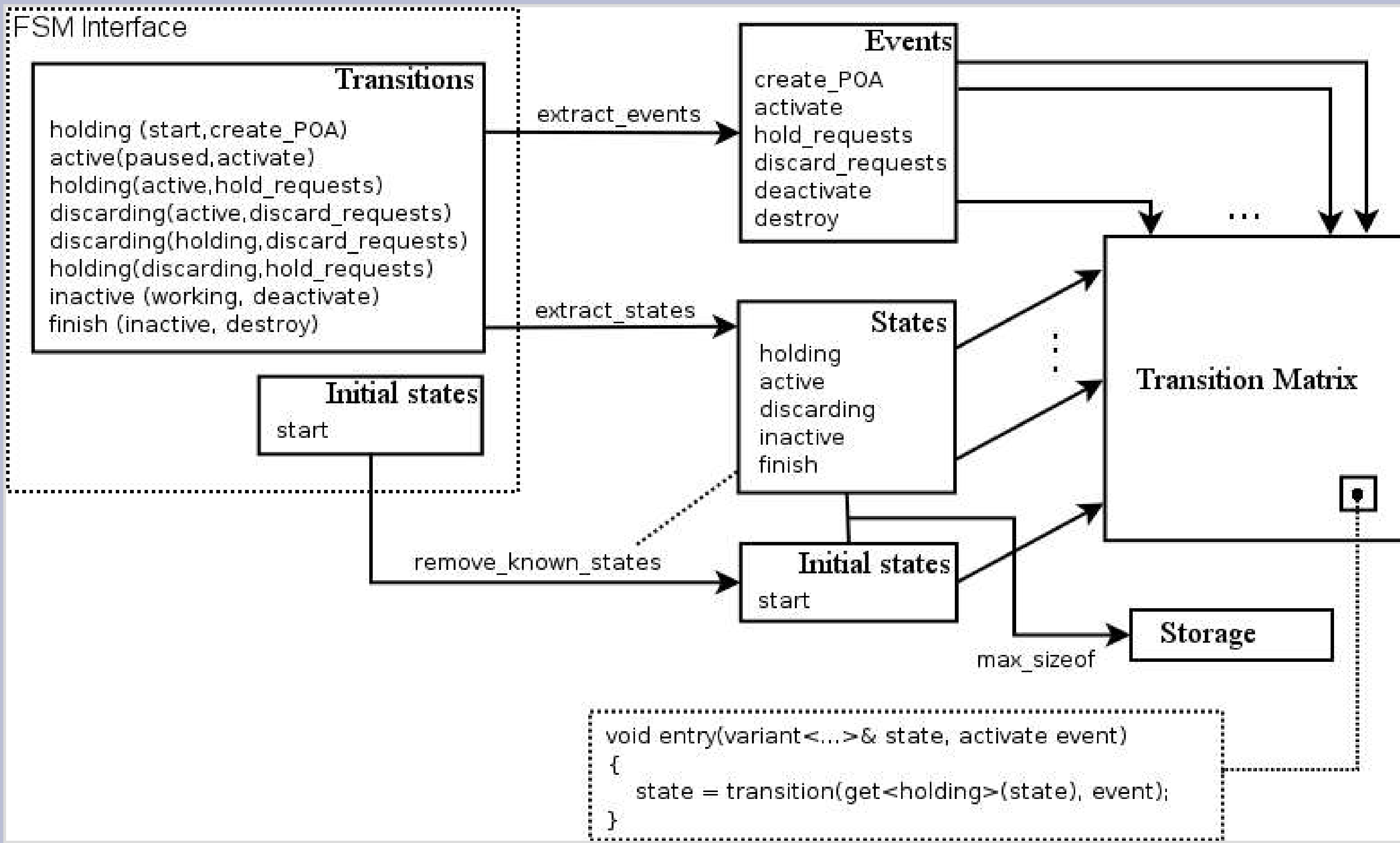
FSM interface

```
int main()
{
    fsm::state_machine<transitions,start_> fsm;

    create_POA_ event;
    fsm.process_event(event);

    if(holding_* pstate = fsm.get_state<holding_>())
    {
        // access pstate data
    }
}
```

Transition in action

```
struct paused_ : working_ {
      // start time of paused states:
      time_t started;
      paused_() : started(time(0)) {}
};

active_ transitions::operator()(
        id<2>, paused_ p, activate_) const
{
     cout << "activate after " << time(0) - p
           << “second(s) pause\n”;
     return active_();
}
```

FSM Road map



FSM Interface
Transitions
holding (start,create_POA)
active(paused,activate)
holding(active,hold_requests)
discarding(active,discard_requests)
discarding(holding,discard_requests)
holding(discarding,hold_requests)
inactive (working, deactivate)
finish (inactive, destroy)
Initial states
start
extract_events
extract_states
Events
create_POA
activate
hold_requests
discard_requests
deactivate
destroy
...
States
holding
active
discarding
inactive
finish
Transition Matrix
remove_known_states
Initial states
start
Storage
max_sizeof
void entry(variant<...>& state, activate event)
{
state = transition(get<holding>(state), event);
}

Getting events out of transitions

```
template<class Signature>
struct get_event
{
    typedef typename remove_cv<
        typename remove_reference<
            typename function_traits<Signature>::arg2_type
          >::type
      >::type type;
};

// extract_events algorithm:
typedef mpl::copy<
    overloads::set<transitions>
  , mpl::inserter<
        mpl::set<>
      , mpl::insert<_1, get_event<_2> >
      >
  >::type events;
```

Getting states out of transitions

```
template<class Signature>
struct get_state
{
    typedef typename remove_cv<
        typename remove_reference<
            typename function_traits<Signature>::result_type
          >::type
      >::type type;
};

// extract_states algorithm:
typedef mpl::copy<
    overloads::set<transitions>
  , mpl::inserter<
        mpl::set<>
      , mpl::insert<_1, get_state<_2> >
      >
  >::type states;
```

remove_known_states algorithm

```
template<class InitialStates, class StateSet>
struct remove_known_states
{
    typedef typename mpl::remove_if<
        InitialStates
      , mpl::has_key<StateSet,_1>
      >::type type;
};

typedef remove_known_states<
    mpl::list<start_>
  , states // defined on previous slide
  >::type initial_states;
```

Transition matrix dimensions

```
typedef typename mpl::size<events>::type event_count;

typedef typename mpl::plus<
    typename mpl::size<states>::type
  , typename mpl::size<initial_states>::type
  >::type state_count;

transition_fn_ptr matrix[event_count::value]
                        [state_count::value];
```

Accessing transition matrix elements

```
// For example: state is holding_, event is activate_

typedef typename mpl::distance<
    typename mpl::begin<events>::type
  , typename mpl::find<events,activate_>::type
  >::type event_index;

// initial_states isn't taken into account,
// it's left as an exercise
typedef typename mpl::distance<
    typename mpl::begin<states>::type
  , typename mpl::find<states,holding_>::type
  >::type state_index;

matrix[event_index::value][state_index::value];
```

Initialization of transition matrix

- Initialize transition matrix with no_transition entries
- Iterate over transitions
- For each transition find all states that can be applied to this transition

  Implemented in **`init_transition`**
- For each such state initialize transition matrix element

  Implemented in **`init_cell`**

Iterate over transitions

```
init_transition init(/* ... */);

mpl::for_each< transitions
               // make a function pointer:
             , add_pointer<_1>
             >(init);
```

init_transition

```
// Template parameters are removed for simplicity
struct init_transition
{
    template<class Sig>
    void operator()(Sig*) const
    {
        typedef typename function_traits<Sig>::arg1_type arg1;

        init_cell<typename get_event<Sig>::type> init(/*...*/);

        mpl::for_each<
            mpl::filter_view<states, is_convertible<_1,arg1> >
          , add_pointer<_1>
          >(init);

        // same action for initial_states ...
    }
};
```

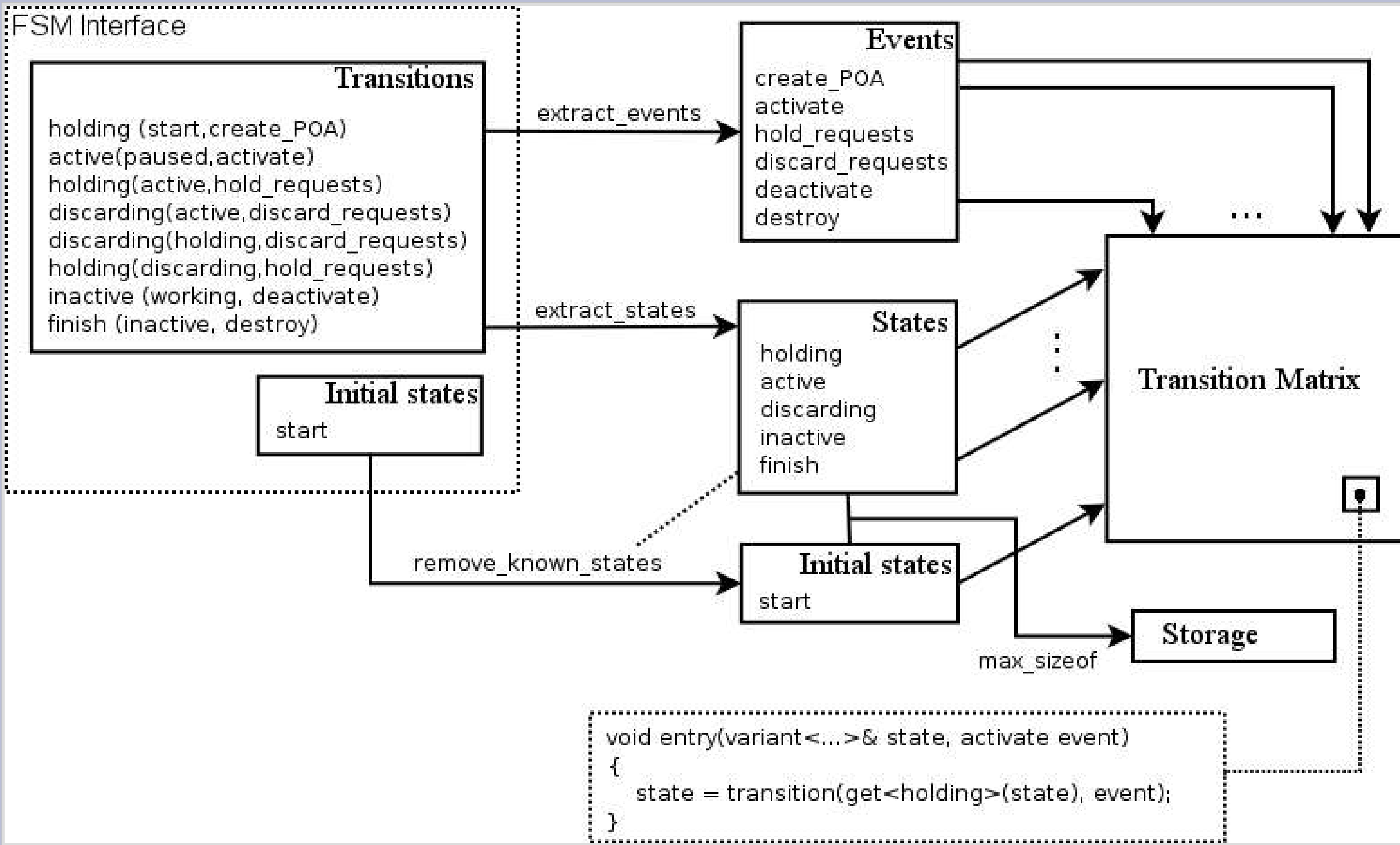
init_cell

```
// Other template parameters are removed for simplicity
template<class Event>
struct init_cell
{
    template<class State>
    void operator()(State*) const
    {
        // event_index and state_index definitions

        matrix[event_index::value][state_index::value] =
            &transition_entry<Event,State,transitions>;
    }
};
```

transition_entry

```
// transition_entry<activate_,holding_,transitions>
void entry( make_variant_over<states>::type& state
          , void const* event
          , transitions const& trans
          )
{
    state = trans( enable_all()
                 , get<holding_>(state)
                 , *static_cast<activate_*>(event)
                 );
}
```

FSM Road map


FSM Interface
Transitions
holding (start,create_POA)
active(paused,activate)
holding(active,hold_requests)
discarding(active,discard_requests)
discarding(holding,discard_requests)
holding(discarding,hold_requests)
inactive (working, deactivate)
finish (inactive, destroy)
Initial states
start
extract_events
extract_states
remove_known_states
Events
create_POA
activate
hold_requests
discard_requests
deactivate
destroy
States
holding
active
discarding
inactive
finish
Initial states
start
Transition Matrix
Storage
max_sizeof
void entry(variant<...>& state, activate event)
{
state = transition(get<holding>(state), event);
}

Disadvantages

- Use of *id<N>* makes it hard to insert/delete an entry from a set, especially from the beginning
  - *Workaround: use constants to start small groups of functions and relative shifts*
  - *Future: get rid of id<N> completely in next revisions of C++ standard*
- Overload set is as extensible as a class
  - *Some tasks require namespace-like extensibility. For example, multiple dispatch.*

Grouping

```
struct transitions
{
    enum { G1 = 0 };
    holding_    operator()(id<G1+1>, start_,      create_POA_      ) const;
    active_     operator()(id<G1+2>, paused_,     activate_        ) const;
    holding_    operator()(id<G1+3>, active_,     hold_requests_   ) const;
    discarding_ operator()(id<G1+4>, active_,     discard_requests_) const;

    enum { G2 = 4 };
    discarding_ operator()(id<G2+1>, holding_,    discard_requests_) const;
    holding_    operator()(id<G2+2>, discarding_, hold_requests_   ) const;
    inactive_   operator()(id<G2+3>, working_,    deactivate_      ) const;
    finish_     operator()(id<G2+4>, inactive_,   destroy_         ) const;
};
```

Minimize compilation time

- Use of typeof where possible
- Special unrolling for sets with fixed arity

```
overloads::set<transitions,fixed_arity>
```

Conceptually, it's an overload set with one very common restriction

FSM-specific optimization

- Put transition table generation into separate translation unit
- Even spread it over several TUs - one TU for one event or small group of events
- Avoid using get_state directly, better get states or results of operations on states through visitors. Put visitation code in a separate TU.

That's it

http://cpp-experiment.sourceforge.net

Questions?