The boost::fsm ライブラリ

チュートリアル(Tutorial)

内容(Contents)

序論(Introduction)

boost::fsmライブラリは、UML状態チャートを実行可能なC++プログラムへ素早く変換することができるフレームワークである。このチュートリアルは、状態機械に関する考え方とUML状態チャートについて、ある程度、精通していることを要求する。状態機械に関する考え方とUML状態チャートの両方についての良い解説が http://www.objectmentor.com/resources/articles/umlfsm.pdfにおいて見つけることができる。状態チャートの考案者であるDavid Harelが、オリジナル論文上において徹底的に行った検討を元にして、チュートリアル的な優れた文書を公開している http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf。UMLに関係した仕様書はhttp://www.omg.org/cgi-bin/doc?formal/03-03-01で見ることがでる(章『2.12』と『3.74』を参照のこと)。

全ての例題プログラムは、MSVC7.1とboost-1.30.2においてテストされた。

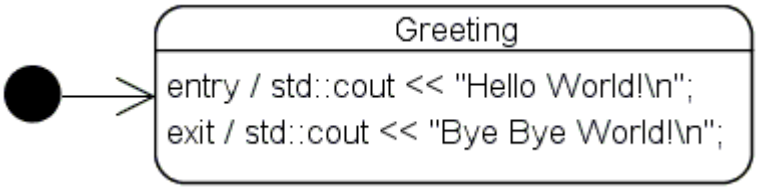
このチュートリアルの読み方（How to read this tutorial）

このチュートリアルは順序よく読まれるようにデザインされている。先ず、利用者は最初から正しく読み始め、目的とした事柄を十分に知ることができた場合、直ちに読むのを止めるようにすべきである。具体的に述べると:

- チュートリアルは、全てのライブラリ利用者が理解するために最も基本的な機能を 説明した『こんにちは世界（Hello World!）』と『ストップウォッチ（stop watch）』例題プログラムで始まる。 扱い易い状態数を持つ小型で単純な状態機械は、基本的な機能を使って無理のない実装を行うことができる。
- 続く『digital camera』例題プログラムでは、誰もが作成しようとする、約1ダース程の状態数を持つ、大きな状態機械による、最も良く知られた機能について説明する
- 最終的に、より複雑な状態機械を作成しようとする利用者とboost::fsmを評価するプロジェクト・アーキテクチャは『より先進的な話題（Advanced topics）』 を最後に読むとよい。加えて、『理論的解釈（Rationale）』 にある『制限（Limitations）』読むことを強く勧める。

こんにちは、世界！（Hello World!）

慣例に従い、最初のステップとして可能限り簡単なプログラムを作成し、それを使用する。次の状態チャートを実装する:




```
#include <boost/fsm/state_machine.hpp>
#include <boost/fsm/simple_state.hpp>
#include <iostream>

namespace fsm = boost::fsm;

struct Greeting;
struct Machine : fsm::state_machine< Machine, Greeting > {};

struct Greeting : fsm::simple_state< Greeting, Machine >
{
  Greeting() { std::cout << "Hello World!\n"; } // entry
  ~Greeting() { std::cout << "Bye Bye World!\n"; } // exit
};

int main()
{
  Machine myMachine;
  myMachine.initiate();
  return 0;
}
```

このプログラムはHello World! と、そして終了する前にBye Bye World!を出力する。最初の行はinitiate()関数を呼び出した結果、Greeting状態が開始されて出力される。main()の終わりでは、myMachineオブジェクトは破棄されGreeting状態から自動的に退場する。

いくつかのの留意点:

- boost::fsmは巧妙に再帰的テンプレートパターンを多用する。派生クラスは、最初のパラメータとして、基底クラステンプレートへ常に渡されなければならない。
- 状態機械は構築されただけでは実行しない。initiate()関数の呼び出しにより実行を開始する。
- 状態機械は、ある状態への入場時には必ず、その状態クラスに対応したオブジェクトを作成する。そのオブジェクトは状態機械がその状態で在る限り生き残り続ける。最終的に、当該オブジェクトは状態機械がその状態から退場した時点で破棄される。それ故、ある状態への入場動作は継いだコンストラクタにより定義可能であり、状態からの退場動作は継ぎ足したデストラクタにより定義することができる。
- 全ての状態はコンテキストに属する。さしあたり、ここでのコンテキストは状態機械のことである。そんな訳で、MachineクラスはGreetingクラスの基底テンプレートクラスに対する二番目のパラメータとして渡される。
- 状態機械は、初期化される時に入場しなければならない状態についての情報を与えられなければならない。それ故Greetingクラスは、Machine基底クラスにおける二番目のテンプレートパラメータとして渡される。このためにGreetingクラスを前方宣言しなければならない。
- public宣言を回避するためだけの目的で、全ての型を構造体(struct)として宣言している。もし、そうする必要がなければ、当然のことながら、クラス(class)の使用は可能である。

ストップウォッチ(A stop watch)

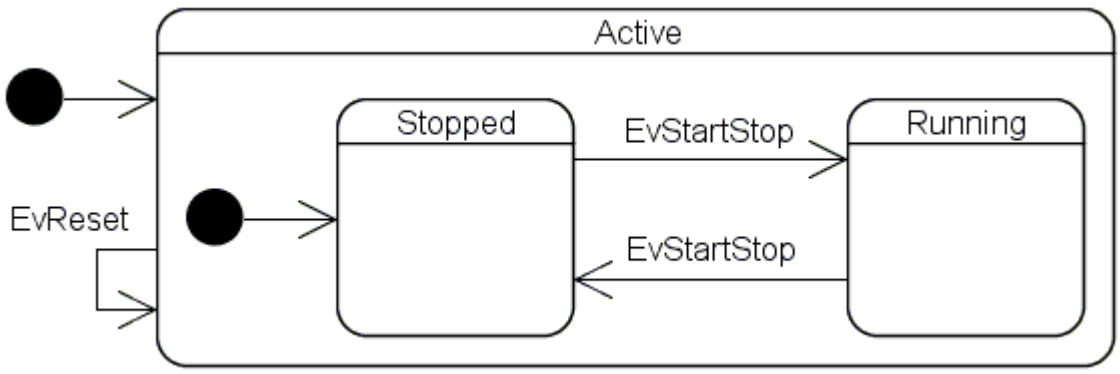
次に、簡単な機械式ストップウォッチを状態機械として形式化する。そのような時計には普通二つのボタンがある:

- 開始/停止ボタン
- リセットボタン

そして二つの状態がある:

- Stopped状態: 針は最後に停止ボタンが押された位置に留まる。
  - リセットボタンを押すと針は0の位置に戻る。時計はStopped状態のままである。
  - 開始/停止ボタンを押すとRunning状態へ遷移する。
- Running状態: 時計の針は作動中となり経過時間を刻み続ける。
  - リセットボタンを押すと針は0の位置に戻り、時計はStopped状態へ遷移する。
  - 開始/停止ボタンを押すとStopped状態へ遷移する。

UMLによりこの状態を示す:



状態とイベントの定義(Defining states and events)

二つのボタンは二つのイベントに形式化される。さらに、必要な状態と初期化用状態を定義する。**次のコードは出発点であり、引き続きコードの断片を挿入していく必要がある**:

```
#include <boost/fsm/event.hpp>
#include <boost/fsm/state_machine.hpp>
#include <boost/fsm/simple_state.hpp>

namespace fsm = boost::fsm;

struct EvStartStop : fsm::event< EvStartStop > {};
struct EvReset : fsm::event< EvReset > {};

struct Active;
struct StopWatch : fsm::state_machine< StopWatch, Active > {};

struct Stopped;
struct Active : fsm::simple_state< Active, StopWatch,
  fsm::no_reactions, Stopped > {};
struct Running : fsm::simple_state< Running, Active > {};
struct Stopped : fsm::simple_state< Stopped, Active > {};

int main()
{
  StopWatch myWatch;
  myWatch.initiate();
  return 0;
}
```

これはコンパイルできるが、まだ見るべきものは何も無い。いくつかのコメントを記述する:

- simple_stateクラステンプレートは四つのパラメータを受け付ける。
  - 三番目のパラメータは反応を明示する(これについては後で説明)。理由は、デフォルトでもあるfsm::no_reactionsを渡しており、まだ語るべきものがないからである。
  - 四番目のパラメータは、もし有るならば、内部の初期化用状態を明示する。
- ある状態は、そのコンテキストである外側の状態を渡すためだけに、内部状態として定義される(最も外側の状態では状態機械を渡す)。
- 状態機械は"外側から内側"に向かって定義する必要があるため、状態のコンテキストは完全な型定義でなければならない(換言すると、前方宣言であってはならない)。これはいつも状態機械から開始し、最も外側の状態が続き、その最外部の内側の状態が次に続き、さらに・・・。幅優先か深さ優先、またはその両者の混合を用いることができる。
  遷移元の状態と遷移先状態はしばしば同じ入れ子の深さとなるため、本当に意味での深さ優先は遷移先の前方宣言を多く定義する傾向があり、幅優先では前方宣言が少なくなる傾向がある。

反応の追加(Adding reactions)

反応はあるイベントの処理過程の結果として起こる何かである。さしあたり、複数の遷移について、一つの型だけによる反応を扱う。以下のコードでは太字部分を**追加**している。

```
#include <boost/fsm/transition.hpp>

// ...

struct Stopped;
```

```
struct Active : fsm::simple_state< Active, StopWatch,
  fsm::transition< EvReset, Active >, Stopped > {};
struct Running : fsm::simple_state< Running, Active,
  fsm::transition< EvStartStop, Stopped > > {};
struct Stopped : fsm::simple_state< Stopped, Active,
  fsm::transition< EvStartStop, Running > > {};

int main()
{
  StopWatch myWatch;
  myWatch.initiate();
  myWatch.process_event( EvStartStop() );
  myWatch.process_event( EvStartStop() );
  myWatch.process_event( EvStartStop() );
  myWatch.process_event( EvReset() );
  return 0;
}
```

状態は任意の数の反応を定義することができる。そのため、反応が1つ以上ある場合は、mpl::list<>内へそれらを設定しなければならない（詳細は、『状態における複数反応の明示（Specifying multiple reactions for a state）』を参照のこと）。
これで、配置する全状態と全ての遷移、そして幾つかのストップウォッチへ通知されるイベントを持つプログラムが出来上がった。状態機械は、期待に添った遷移を従順に遂行できるが、動作はまだ実行されない。

状態 - ローカル記憶領域（State-local storage）

次はストップウォッチに対し、実際に時間を測定させてみる。ストップウォッチの状態によって、異なる変数が必要になる：

- Stopped状態：経過時間を保持するひとつの変数
- Running状態：経過時間を保持するひとつの変数**と**最後に開始した時点の時刻を保持しておく一変数

経過時間変数は、状態機械の状態に係わらず必要となることに気が付く。そのうえ、この変数はEvResetイベントを状態機械に発行したとき、0にリセットされる。他の変数は、この状態機械がRunning状態の間だけ必要とされる。それへは、Running状態へ入る度に、システムクロックの現在時刻がセットされる。Running状態から出る時、現在のシステム時刻からスタート時刻を単に減算し、結果を経過時間へ加算する。

```
#include <ctime>

// ...

struct Stopped;
struct Active : fsm::simple_state< Active, StopWatch,
  fsm::transition< EvReset, Active >, Stopped >
{
  public:
    Active() : elapsedTime_( 0 ) {}
    std::clock_t ElapsedTime() const { return elapsedTime_; }
    std::clock_t & ElapsedTime() { return elapsedTime_; }
  private:
```

```
    std::clock_t elapsedTime_;
};

struct Running : fsm::simple_state< Running, Active,
  fsm::transition< EvStartStop, Stopped > >
{
  public:
    Running() : startTime_( std::clock() ) {}
    ~Running()
    {
      context< Active >().ElapsedTime() +=
        ( std::clock() - startTime_ );
    }
  private:
    std::clock_t startTime_;
};

// ...
```

派生したクラスが基本クラス部分へアクセスするのと同様に、context<>()関数は直接または間接的に外側の状態を入手するために使用される。同じ関数は状態機械を入手するためにも使われる(ここでは、context< StopWatch >()関数)。残りの大部分のコードは自明である。現在、状態機械は時間を計れるが、メイン処理からは未だそれを取得できない。

状態機械外からの状態の情報の取得(Getting state information out of the machine)

測定時間を取得するには、状態機械の外側から状態の情報を取り出す仕組みが必要である。現状の状態機械のデザインでは、そのためにふたつの手段がある。簡単にするために、それほど効率的でないstate_cast<>()を利用する。名前が示唆する通り、意味はdynamic_castのそれと類似している。例えば、myWatch.state_cast< const Stopped & >()を呼び、**かつ**状態機械が現在Stopped状態である場合に、Stopped状態への参照を得る。その他の場合は、std::bad_castが送出される。経過時間を戻すためのStopWatchメンバー関数を実装する場合にこの機能を利用することができる。しかしながら、状態機械がどんな状態かを問い、それから経過時間のために各種の計算を切り替えるよりは、むしろStopped状態とRunning状態へ計算処理を埋め込み、経過時間を取得するためのインターフェースとして利用するようにする:

```
#include <iostream>

// ...

struct IElapsedTime
{
  virtual std::clock_t ElapsedTime() const = 0;
};

struct Active;
struct StopWatch : fsm::state_machine< StopWatch, Active >
{
  std::clock_t ElapsedTime() const
  {
    return state_cast< const IElapsedTime & >().ElapsedTime();
  }
```

```
};

// ...

struct Running : IElapsedTime, fsm::simple_state<
  Running, Active, fsm::transition< EvStartStop, Stopped > >
{
  public:
    Running() : startTime_( std::clock() ) {}
    ~Running()
    {
      context< Active >().ElapsedTime() = ElapsedTime();
    }

    virtual std::clock_t ElapsedTime() const
    {
      return context< Active >().ElapsedTime() +
        std::clock() - startTime_;
    }
  private:
    std::clock_t startTime_;
};

struct Stopped : IElapsedTime, fsm::simple_state<
  Stopped, Active, fsm::transition< EvStartStop, Running > >
{
  virtual std::clock_t ElapsedTime() const
  {
    return context< Active >().ElapsedTime();
  }
};

int main()
{
  StopWatch myWatch;
  myWatch.initiate();
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvStartStop() );
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvStartStop() );
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvStartStop() );
  std::cout << myWatch.ElapsedTime() << "\n";
  myWatch.process_event( EvReset() );
  std::cout << myWatch.ElapsedTime() << "\n";
  return 0;
}
```

実際に測定された時間を見るために、main()関数におけるステップ毎の実行文が必要となる。
StopWatch例題プログラムは、このプログラムを会話型のコンソールアプリケーションに拡張している。

デジタルカメラ(A digital camera)

いまのところ順調である。しかしながら、ここまでの取り組みには幾つかの制約がある:

- 拡張性の悪さ: コンパイラは、state_machine::initiate()が呼び出さる箇所に到達した時点で幾つかのテンプレートの実体化が起こるが、これは状態機械が知る全ての状態と各状態が完全に定義された場合にのみ成功する。状態機械の全レイアウトは、一回の翻訳単位として実装されなければならない(動作は分割してコンパイルできるが、これはここではあまり重要ではない)。大きな(かつ現実の世界における)状態機械では、以下の制約を課す:
  - いくつかのコンパイラは、テンプレート実体化の内部処理過程における限界に突き当たり、これを放棄する。このことは、適切と思われる大きさの状態機械でも起き得る。例えば、或る一般的なコンパイラによるデバックモードでは、3ビット以上を定義したBitMachine例題プログラムの初期バージョンのコンパイルを拒絶した。これは、8状態／24遷移数から16状態／64遷移数の間でコンパイラが限界に到達することを意味する。
  - 複数のプログラマは、同一の状態機械上で厳密な仕事が同時にできる、なぜなら全てのレイアウト変更は、必然的に全状態機械の再コンパイルを引き起こすからである。
- イベント毎に最大ひとつの反応: UMLによれば、ある状態は同じイベントにより引き起こされる複数の反応を持てる。これは、全ての反応は相互に排他的なガードを持つことを意味する。これまでに利用したインターフェースは、各イベント毎に最大ひとつのガード無しの反応を認めている。そのうえ、UMLにおける接合と選択ポイントの概念は、直接サポートされない。
- 状態内の反応については方法を明確にしていない(内部の遷移についても同様)。

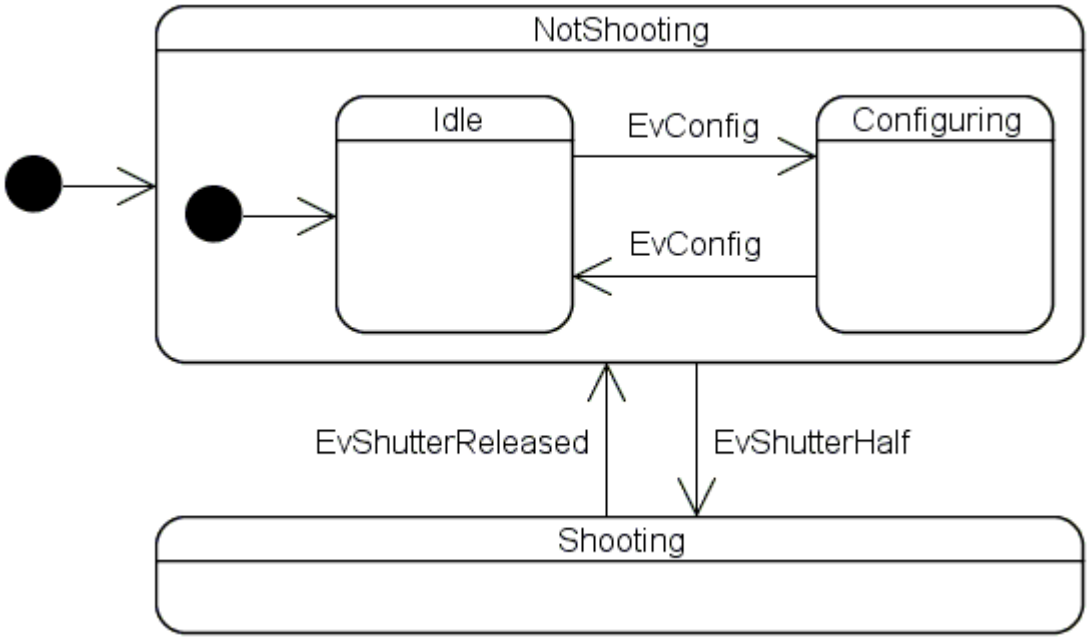
これらの全ての制限は所望する反応により克服することができる。**警告: 所望する反応の安易な乱用は未定義な振る舞いを引き起こす。それらを使用する前にこのドキュメントで学習すること!**

状態機械の複数遷移ユニットへの拡張(Spreading a state machine over multiple translation units)

あなたの会社がデジタルカメラを開発すると仮定する。そのカメラは以下の制御を行う:

- シャッターボタン、これには半押しと全押しがある。それに関連付けられたイベント として、`EvShutterHalf`と`EvShutterFull`、`EvShutterReleased`がある
- 構成ボタン、`EvConfig`イベントによって表される
- ここでは、他のボタンについて関心がない

カメラを使う場合に言える事は、撮影者は構成モードの**何処**にいてもシャッターを半押しすることができ、カメラは直ちに撮影モードになる。次の状態チャートはこの振舞いを達成する方法を示す:



Configuring状態とShooting状態は多数の入れ子の状態を抱合しているが、Idle状態は割と単純である。それは、二つの組として形成することを決定付ける。一方は撮影モードとして実装し、もう片方は構成モードとして実装する。撮影モードが設定された構成情報を取得するために利用するインターフェースについて、二つのモードでは既に一致している。二つのモードが出来る限り干渉せずに動作できることを保証したい。そこで、Configuring状態に関する状態機械の構造変更がShooting状態内を再コンパイルする作業に係わらず、またその逆についても言えるように、ふたつの状態を各々の翻訳単位に分ける。

前の例とは異なり、抜粋箇所は、同じ結果に到達する為の異なる選択肢となる輪郭をここでは表している。それ故、このコードはCamera例題プログラムのコードと所々で等しくない。 このケースに当て嵌まる部分をコメントとして印している。

Camera.hpp:

```
#ifndef CAMERA_HPP
#define CAMERA_HPP

#include <boost/fsm/event.hpp>
#include <boost/fsm/state_machine.hpp>
#include <boost/fsm/simple_state.hpp>
#include <boost/fsm/custom_reaction.hpp>

namespace fsm = boost::fsm;

struct EvShutterHalf : fsm::event< EvShutterHalf > {};
struct EvShutterFull : fsm::event< EvShutterFull > {};
struct EvShutterRelease : fsm::event< EvShutterRelease > {};
struct EvConfig : fsm::event< EvConfig > {};

struct NotShooting;
struct Camera : fsm::state_machine< Camera, NotShooting >
{
  bool IsMemoryAvailable() const { return true; }
  bool IsBatteryLow() const { return false; }
};
```

```
struct Idle;
struct NotShooting : fsm::simple_state< NotShooting, Camera,
  fsm::custom_reaction< EvShutterHalf >, Idle >
{
  // ...
  fsm::result react( const EvShutterHalf & );
};

struct Idle : fsm::simple_state< Idle, NotShooting,
  fsm::custom_reaction< EvConfig > >
{
  // ...
  fsm::result react( const EvConfig & );
};

#endif
```

コードにおける太字部分に注目してください。所望の反応に関して、特定のイベントに対して何かすることを示唆するが、実際の反応はreactメンバー関数として定義され、.cppファイルに実装される。

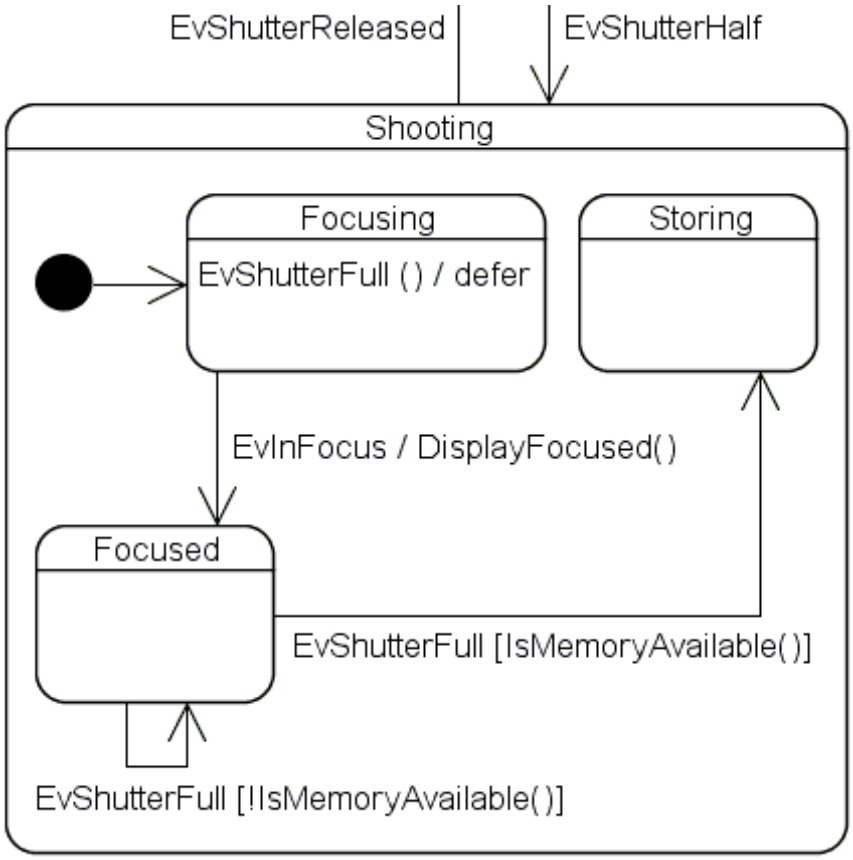
Camera.cpp:

```
#include "Camera.hpp"
#include "Configuring.hpp"
#include "Shooting.hpp"

// ...

// not part of the Camera example
fsm::result NotShooting::react( const EvShutterHalf & )
{
  return transit< Shooting >();
}

fsm::result Idle::react( const EvConfig & )
{
  return transit< Configuring >();
}
```

注意事項: simple_state::transit<>() または simple_state::terminate() (『反応関数の参照(Reaction function reference)』を見よ)メンバー関数を呼び出すことは、現状態オブジェクトの破棄(delete this;と同義)を必然的に起こす。このことは、これらの呼び出し後のコードの実行は未定義の振舞いを引き起こすだろう。 それ故、これらの関数はリターン文の一部としてだけ呼び出されるようにする。

防御(Guards)

Shooting状態の内部の動きは次のように見える:



利用者がシャッターを半押しとしたとき、Shooting状態と内部の初期化のためにFocusing状態に入る。カメラのFocusing状態に入る動作は、焦点回路に焦点を対象へ合せるように指示する。その時、焦点回路は焦点を合せるのためにレンズを動かし、合致すると同時にEvInFocusイベントを送る。もちろん、まだレンズが動いている間でも、利用者はシャッターを全押しにできる。 Focusing状態はEvShutterFullイベントに対する反応を定義していないため、何らかの事前警告なしのEvShutterFullイベントは単に失われるだけである。結果として、カメラが焦点を定めた後、利用者は再びシャッターを全押ししなければならない。これを防止するために、EvShutterFullイベントはFocusing状態内で遅延される。このことは、分離したキューに保持されたこのタイプの全てのイベントは、 Focusing状態から出る時にメインキューへ移されることを意味する。

```
struct Focusing : fsm::state< Focusing, Shooting, mpl::list<
  fsm::custom_reaction< EvInFocus >,
  fsm::deferral< EvShutterFull > > >
{
  Focusing( my_context ctx );
  fsm::result react( const EvInFocus & );
};
```

Focused状態を起点とする両方の遷移は同じイベントにより引き起こされるが、相互に排他的な監視行っている。ここでは、適切な反応の例を示す。

```
// not part of the Camera example
fsm::result Focused::react( const EvShutterFull & )
{
  if ( context< Camera >().IsMemoryAvailable() )
  {
    return transit< Storing >();
  }
  else
```

```
    {
      // 次は実際に状態内反応と遷移とを合成したものである。
      // 後方の適切な遷移動作の実装方法を見よ。
      std::cout << "Cache memory full. Please wait...\n";
      return transit< Focused >();
    }
  }
```

所望の反応は、当然のことながら状態宣言内へ直接記述することが可能であり、そのためコードを容易に見渡すことが出来るので好ましい。

次に電池が少なくなった場合に備え、遷移防止のために監視機能を利用すると共にイベントを介して外側の状態を動作させるようにする:

Camera.cpp:

```
// ...
fsm::result NotShooting::react( const EvShutterHalf & )
{
  if ( context< Camera >().IsBatteryLow() )
  {
    // 自身ではイベントに反応することはできないので、外側の状態へ転送する
    // （これはまたイベントに対する反応処理を定義しなかった場合の
    // デフォルト処理である）
    return forward_event();
  }
  else
  {
    return transit< Shooting >();
  }
}
// ...
```

状態内の反応(In-state reactions - または内部遷移(aka inner transitions))

Focused状態の自身への遷移は内部状態反応として実装され、Focused状態が他の入場または退場動作を持たない限り同じ効果となる:

Shooting.cpp:

```
// ...
fsm::result Focused::react( const EvShutterFull & )
{
  if ( context< Camera >().IsMemoryAvailable() )
  {
    return transit< Storing >();
  }
  else
  {
    std::cout << "Cache memory full. Please wait...\n";
    // イベントは処分されることを示す。
    // そこでディスパッチアルゴリズムは反応を探すことを止め、
    // 状態機械はFocused状態に留まる。
```

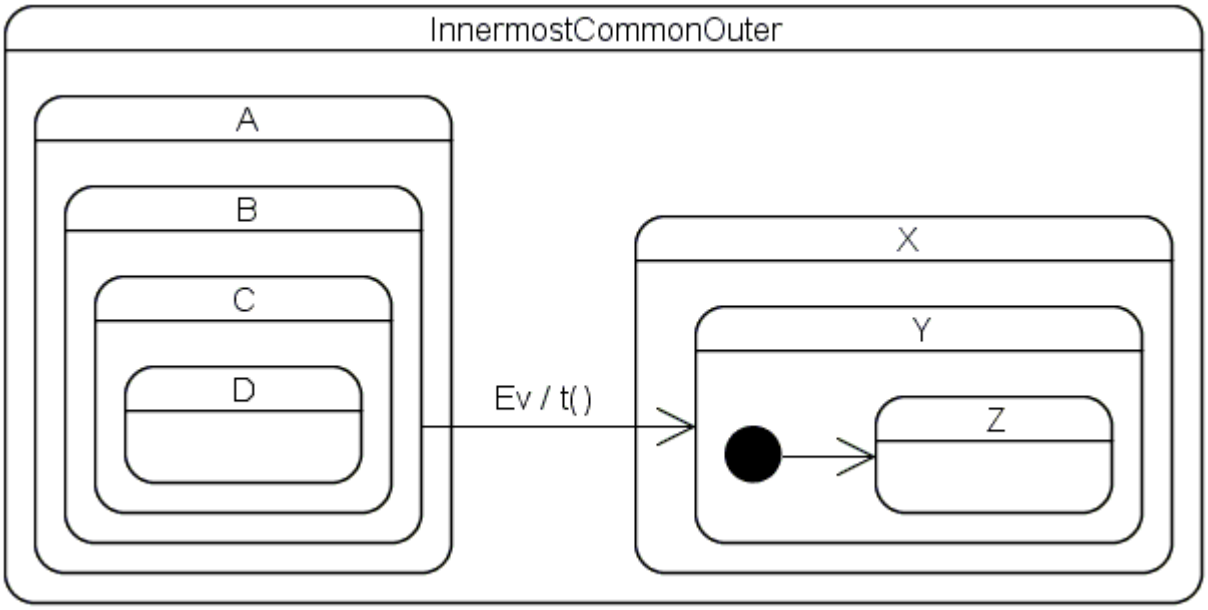
```
      return discard_event();
    }
  }
  // ...
```

遷移動作(Transition actions)

全ての遷移の作用として、動作は次の順序で実行される:

1. 最深部の現状態から開始し、全ての退場動作を遡るが、最深部に共通の外部状態は除外する。(LCA:least common ancestor:最も近い共通の先祖と同義)
2. 遷移動作(もし在れば)。
3. 内側の最も共通な外部状態から開始して、全ての入場動作は、初期化用状態の入場動作によって、次の対象となる状態に向かって下降する。

例:



ここでの順序は次の通り:~D(), ~C(), ~B(), ~A(), t(), X(), Y(), Z() 。それ故、遷移動作 t()がInnermostCommonOuter状態のコンテキストにおいて実行されるのは、元の状態が退場(破棄)されている上に、対象とされる状態は未だ入場(生成)していないからである。

boost::fsmに関して、遷移動作は、**どのような**共通の外部コンテキストのメンバーにもなることができる。そのため、Focusing状態とFocused状態の間の遷移は次のように実装される:

Shooting.hpp:

```
  // ...
  struct Focusing;
  struct Shooting : fsm::simple_state< Shooting, Camera,
    fsm::transition< EvShutterRelease, NotShooting >, Focusing >
  {
    // ...
    void DisplayFocused( const EvInFocus & );
  };
```

```
// ...

// not part of the Camera example
struct Focusing : fsm::simple_state< Focusing, Shooting,
  fsm::transition< EvInFocus, Focused,
    Shooting, &Shooting::DisplayFocused > > {};
```

または、次も可能である(ここでの状態機械は最外部のコンテキストとして自身を配布する):

```
// not part of the Camera example
struct Camera : fsm::state_machine< Camera, NotShooting >
{
  void DisplayFocused( const EvInFocus & );
};

// not part of the Camera example
struct Focusing : fsm::simple_state< Focusing, Shooting,
  fsm::transition< EvInFocus, Focused,
    Camera, &Camera::DisplayFocused > > {};
```

普通、遷移動作は所望の反応から引き起こされる:

Shooting.cpp:

```
// ...
fsm::result Focusing::react( const EvInFocus & evt )
{
  return transit< Focused >( &Shooting::DisplayFocused, evt );
}
```

手動でイベントを転送しなければならないことに注意すること。

より先進的な話題(Advanced topics)

反応機能の参照(Reaction function reference)

次の関数はreactメンバー関数から単独で呼び出すことができるが、決められた関数の呼び出しにより戻らなければならない(例えば、`return terminate();`):

- `simple_state::forward_event()`: ディスパッチアルゴリズムは現イベントのために反応を検索する情報を保持する。検索は常に隣接する外部状態により行われる。もし隣接する状態が無ければ、次に直交する末端状態に対してそれを続ける。この処理は、他の五つの反応関数のうちのいずれかの呼び出しによって、訪れた状態のひとつが戻るまで繰り返される。もし反応が見つけることが出来なければ、イベントは暗黙の内に処分される。これは、監視機能の実装に役に立つ。`forward_event()`関数もまた、全ての状態におけるデフォルト処理であり、イベントに対する反応を定義していない。
- `simple_state::discard_event()`: このディスパッチアルゴリズムは反応の検索を止め、そして現イベントは処分される。これは状態内の反応の実装に役に立つ。
- `simple_state::defer_event()`: 現イベントは分離キューへプッシュされ、ディスパッチアルゴリズムは反応の検索を止める。状態が退場させられた後、分離キューはメインキューに移され、その後は通常の処理となる。『委譲イベント(Deferring events)』を参照のこと。
- `simple_state::transit< DestinationState >()`: 明示された対象状態への遷移処理

の作成と現イベントの廃棄を行なう。
- `simple_state::transit< DestinationState >( void ( TransitionContext::* )( const Event & ), const Event & )`: 通知された遷移動作が呼ばれた間、明示された対象状態への遷移処理の作成と、現イベントの破棄。
- `simple_state::terminate()`: 状態の終了と現イベントの廃棄。

反応の参照(Reaction reference)

`custom_reaction` 以外の反応は無いが、構文にゆとりを持たせているので通常のケースでは、`react`メンバー関数を必ずしも書く必要は無い。ここでは現在提供している反応のリストを示す:

- `transition< Event, DestinationState >`: `simple_state::transit< DestinationState >();`からの戻り。
- `transition< Event, DestinationState, TransitionContext, void ( TransitionContext::*pTransitionAction )( const Event & ) >`: `simple_state::transit< DestinationState >( pTransitionAction, evt );` からの戻り。
- `termination< Event >`: `simple_state::terminate();`からの戻り。
- `deferral< Event >`: `simple_state::defer_event();`からの戻り。『 委譲イベント(Deferring events)』を参照のこと。
- `custom_reaction< Event >`: `react( evt );` からの戻り(利用者が供給するメンバー関数)。`react` メンバー関数は反 応関数のいずれかの呼び出しにより戻らなければならない。

利用者はしばしば彼・彼女自身が実装するものに類似した反応メンバー関数を見つけので、彼・彼女は自身の反応を簡単に定義でき、boost::fsmによって提供されるそれをそのまま利用できる。

状態における複数反応の明示(Specifying multiple reactions for a state)

時々状態は、ひとつ以上のイベント用反応を定義しなければならない。この場合、次のような外枠を示すための`mpl::list`を使用しなければならない:

```
// ...

#include <boost/mpl/list.hpp>

namespace mpl = boost::mpl;

// ...

struct Playing : fsm::simple_state< Playing, Mp3Player,
  mpl::list<
    fsm::custom_reaction< EvFastForward >,
    fsm::transition< EvStop, Stopped > > > { /* ... */ };
```

イベントの通知(Posting events)

特殊な状態機械は内部イベントをしばしば通知する必要がある。ここでは、どのようにこれを行うか例示する:

```
Pumping::~Pumping()
{
  post_event( boost::intrusive_ptr< EvPumpingFinished >(
```

```
      new EvPumpingFinished() ) );
  }
```

イベントはメインキューに登録されるため、new関数により割り当てられなければならない。キュー内のイベントは、現反応の完了後、速やかに処理される。イベントは、entry-、exit-と遷移動作など内部のreact関数から通知される。しかしながら入場動作内からの通知は少し複雑である(Camera例題プログラムにおけるShooting.cppのFocusing::Focusingを参照のこと):

```
  struct Pumping : fsm::state< Pumping, Purifier >
  {
    Pumping( my_context ctx ) : my_base( ctx )
    {
      post_event( boost::intrusive_ptr< EvPumpingStarted >(
        new EvPumpingStarted() ) );
    }
    // ...
  };
```

太字の部分に注意すること。状態の入場動作が直に"外側の世界"(ここでは状態機械内のイベントキュー)と連絡をとる必要が場合、状態はfsm::simple_stateよりむしろfsm::stateから派生しなければならず、上記の概略で述べたように転送用コンストラクタを実装しなければならない(コンストラクタは別にして、fsm::stateはfsm::simple_stateと同じインターフェースを提供する)。そのため、入場動作が次に示す一つまたはそれ以上の関数を呼び出す時はいつでも、そうしなければならない:

- simple_state::context<>()
- simple_state::post_event()
- simple_state::state_cast<>()
- simple_state::state_downcast<>()

私の経験では、これらの関数は入場動作内では稀にしか必要とされないので、この不可欠な作業は利用者のコードをそれ程不恰好にしない。

委譲イベント(Deferring events)

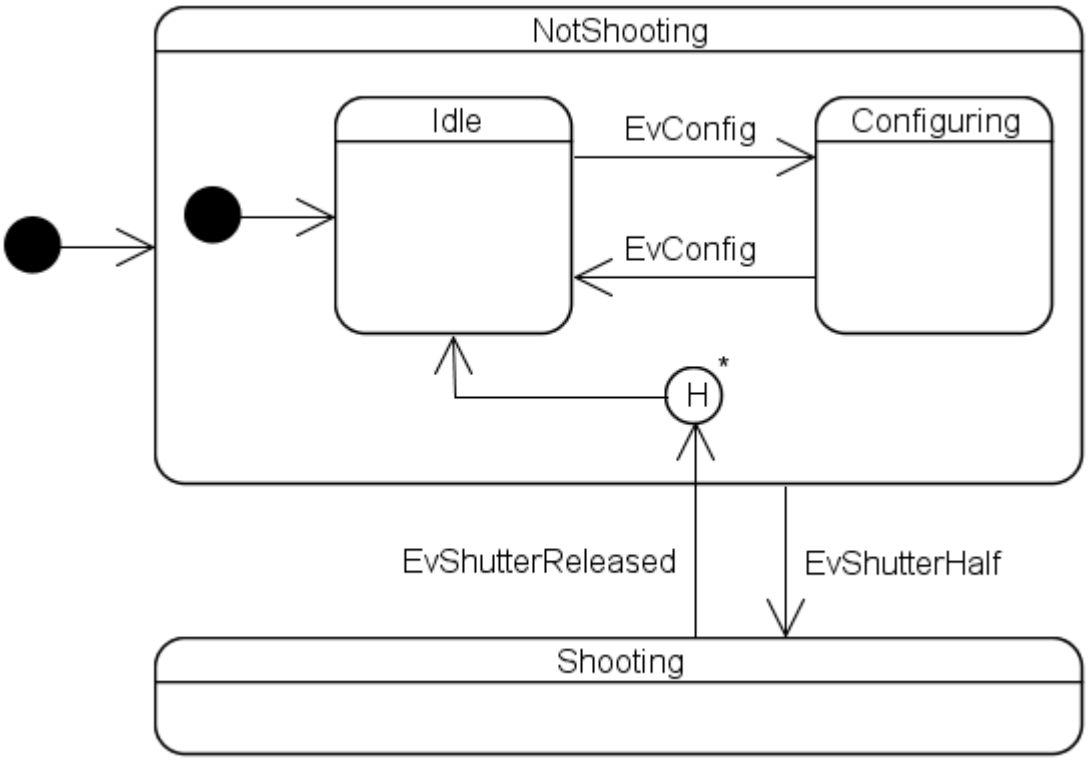
多くのオーバーヘッドを避けるためにイベントの委譲には一つの制限がある:イベントはnew関数により唯一割り当てられる**と同時に**委譲可能な boost::intrusive_ptr<>により指し示される。委譲とは異なった方法で割り当てられたイベントを使用しようとするどのような試みも表明による実行時の失敗となる。例えば:

```
  struct Event : fsm::event< Event > {};
  struct Initial;
  struct Machine : fsm::state_machine<
    Machine, Initial > {};
  struct Initial : fsm::simple_state< Initial, Machine,
    fsm::deferral< Event > > {};

  int main()
  {
    Machine myMachine;
    myMachine.initiate();
    myMachine.process_event( Event() ); // error
    myMachine.process_event(
      *boost::shared_ptr< Event >( new Event() ) ); // error
```

```
    myMachine.process_event(
      *boost::intrusive_ptr< Event >( new Event() ) ); // fine
    return 0;
  }
```

履歴(History)

この デジタルカメラのベータ版を試した撮影者が、シャッターの半押し操作はどんな時でも(カメラの構成中であっても)直ちに撮影可能状態となることが実際ところ好ましいと言った。しかしながら、彼らの多くは、いつもシャッターを解放した後、カメラがIdleモードに入ってしまうことが直感的でないことに気が付く。彼らは、カメラがシャッターを半押しする前の状態に戻ることを、むしろ予想しているようである。このように、撮影するためにシャッターを半押しさらには全押しすることにより、修正した構成情報の設定についての影響を簡単にテストできる。最後に、シャッターの解放は修正した設定内容をスクリーンに反映する。次の様に状態チャートを変更し、この振る舞いを実装する:



以前述べたようにConfiguring状態は少し複雑で、深い入れ子の内部状態機械を抱合している。当然、以前の状態をたどり、Configuring状態における最も深い状態を復元したいために深い履歴擬似状態を利用する。これに関連したコードを次に見る:

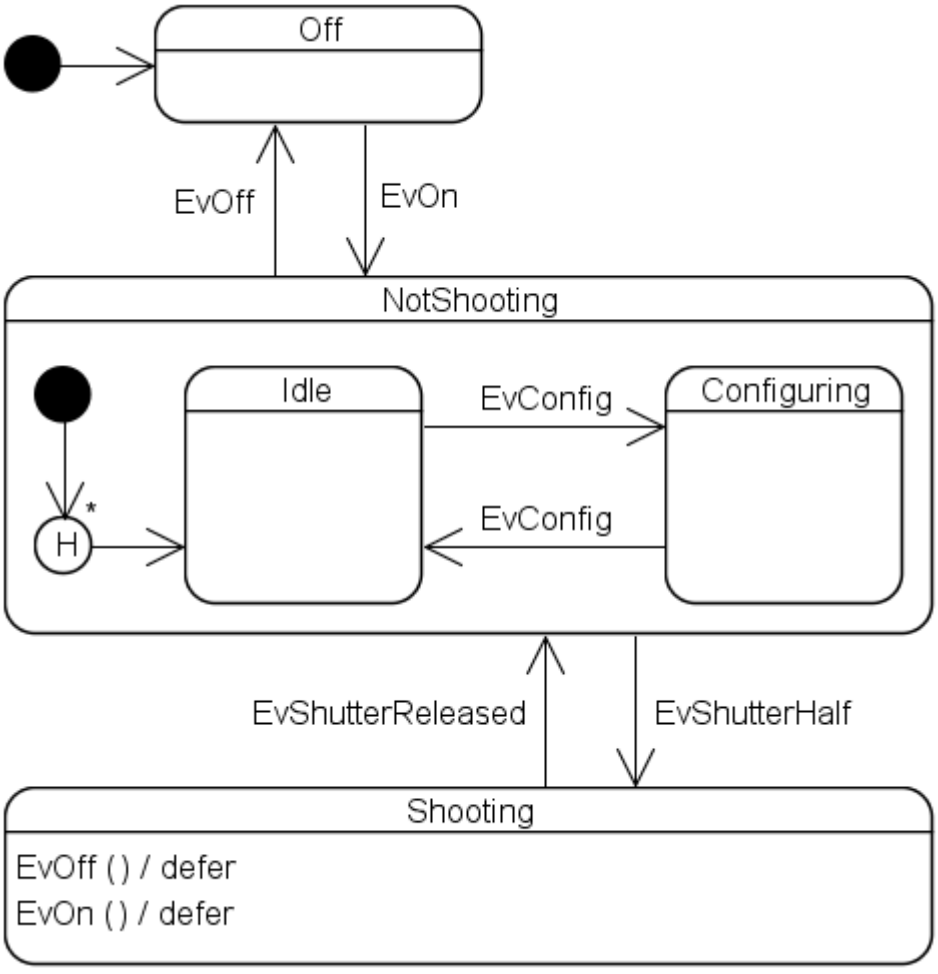
```
  // not part of the Camera example
  struct NotShooting : fsm::simple_state< NotShooting, Camera,
    /* ... */, Idle, fsm::has_deep_history > //
  {
    // ...
  };

  // ...

  struct Shooting : fsm::simple_state< Shooting, Camera,
    fsm::transition< EvShutterRelease,
      fsm::deep_history< Idle > >, Focusing >
```

```
{
  // ...
};
```

状態履歴は二つの相を持つ。第一の相は、履歴擬似状態を含む状態から退場するとき、その前に実行中の内部状態は履歴として保存されなければならない。第二は、履歴擬似状態への遷移が後で作成される時、保存された履歴状態情報は検索され該当する状態として遷移させられる。前者は、simple_stateまたはstateテンプレートの最後のパラメータとして、fsm::has_shallow_history、fsm::has_deep_historyか fsm::has_full_history(浅い履歴と深い履歴が結合したもの)のいずれかを渡すことで表現される。後者は、遷移先としてか、瞬間的に見えるか、内部の初期化用状態としてなのかを fsm::shallow_history<>または fsm::deep_history<> のいずれかを明示することにより表される。なぜなら、履歴擬似状態を含む状態が遷移以前に作成された履歴へ決して入場しないため、両方のテンプレートは、そのような状況における、デフォルトの状態を明確に記述したパラメータを必要とするからである。

履歴を利用するのに必要な冗長部分は、コンパイル時に一貫性がチェックされる。つまり、NotShootingの基底クラスへfsm::has_deep_historyを渡すことを忘れるなどすると状態機械がコンパイルされないためである。

もうひとつの要求される変更は、幾人かのベータテスタによって、カメラの状態を元に戻した時に、状態が切り替る前にカメラの状態が戻るように見えたと提起されたことによる。これがその実装である:

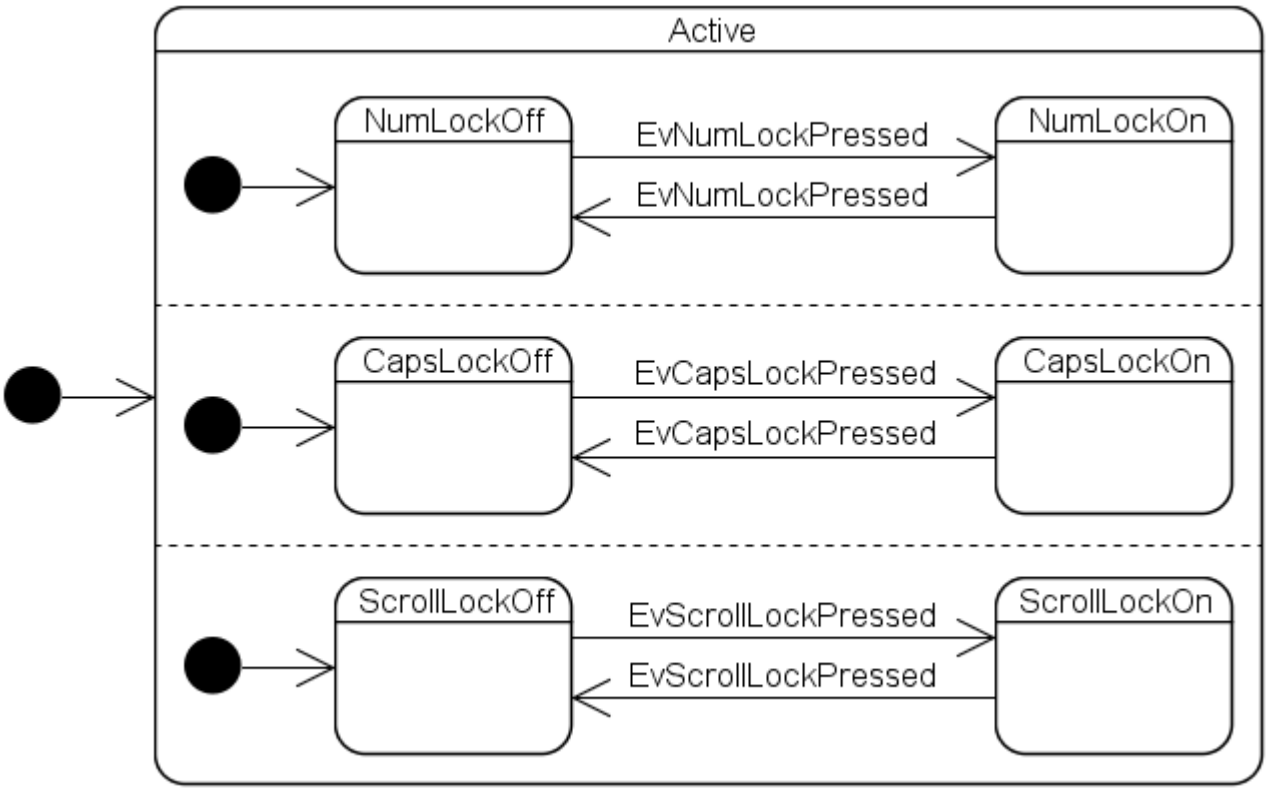



```
// ...

// not part of the Camera example
struct NotShooting : fsm::simple_state< NotShooting, Camera,
```

```
  /* ... */, mpl::list< fsm::deep_history< Idle > >,
  fsm::has_deep_history >
{
  // ...
};

// ...
```

あいにく、幾つかのテンプレートに関連した詳細な実装の結果、少し不都合がある。内部の初期化用状態が一つだけであっても、常にクラステンプレートの具象化したクラスを mpl::list<>に設定しなければならない。さらに、現在の深い履歴の実装は幾つかの制限がある。

直交状態(Orthogonal states)



この状態チャートを実装するために、ひとつ以上の内部初期状態を明示的に記述しなければならない(Keyboard例題プログラムを参照のこと):

```
struct Active;
struct Keyboard : fsm::state_machine< Keyboard, Active > {};

struct NumLockOff;
struct CapsLockOff;
struct ScrollLockOff;
struct Active: fsm::simple_state<
  Active, Keyboard, fsm::no_reactions,
  mpl::list< NumLockOff, CapsLockOff, ScrollLockOff > > {};
```

Active状態の内部状態は、それらが属している直交領域を定義しなければならない:

```
struct EvNumLockPressed : fsm::event< EvNumLockPressed > {};
```

```
struct EvCapsLockPressed : fsm::event< EvCapsLockPressed > {};
struct EvScrollLockPressed :
  fsm::event< EvScrollLockPressed > {};

struct NumLockOn : fsm::simple_state<
  NumLockOn, Active::orthogonal< 0 >,
  fsm::transition< EvNumLockPressed, NumLockOff > > {};
struct NumLockOff : fsm::simple_state<
  NumLockOff, Active::orthogonal< 0 >,
  fsm::transition< EvNumLockPressed, NumLockOn > > {};

struct CapsLockOn : fsm::simple_state<
  CapsLockOn, Active::orthogonal< 1 >,
  fsm::transition< EvCapsLockPressed, CapsLockOff > > {};
struct CapsLockOff : fsm::simple_state<
  CapsLockOff, Active::orthogonal< 1 >,
  fsm::transition< EvCapsLockPressed, CapsLockOn > > {};

struct ScrollLockOn : fsm::simple_state<
  ScrollLockOn, Active::orthogonal< 2 >,
  fsm::transition< EvScrollLockPressed, ScrollLockOff > > {};
struct ScrollLockOff : fsm::simple_state<
  ScrollLockOff, Active::orthogonal< 2 >,
  fsm::transition< EvScrollLockPressed, ScrollLockOn > > {};
```

orthogonal< 0 > は省略値であり、NumLockOn状態とNumLockOff状態は、それらの明示しているコンテキストへActive::orthogonal< 0 >の代わりとして、Active状態を同様に渡すことができる。orthogonalメンバーテンプレートへ渡す番号は、外側の状態におけるリスト位置と一致しなければならない。さらに、遷移元状態の直交位置は遷移先状態の位置と一致する必要がある。これらのルールに対する何らかの違反は、コンパイルエラーをもたらす。例えば:

```
// 例１：有効な直交領域は三個だけなのでコンパイルできない。
struct WhateverLockOn: fsm::simple_state<
  WhateverLockOn, Active::orthogonal< 3 > > {};

// 例２：NumLockOff 状態は"０番目"の直交領域なのでコンパイルできない。
struct NumLockOff : fsm::simple_state<
  NumLockOff, Active::orthogonal< 1 > > {};

// 例３：異なったの直交領域間での遷移は認めていないのでコンパイルできない。
struct CapsLockOn : fsm::simple_state<
  CapsLockOn, Active::orthogonal< 1 >,
  fsm::transition< EvCapsLockPressed, CapsLockOff > > {};
struct CapsLockOff : fsm::simple_state<
  CapsLockOff, Active::orthogonal< 2 >,
  fsm::transition< EvCapsLockPressed, CapsLockOn > > {};
```

状態の問い合せ（State queries）

しばしば、状態機械における反応は、ひとつまたはそれ以上の直交領域内の現状態に依存する。換言すると、直交領域は完全に直交していないか、内部の直交領域が特別な状態の場合だけ起こりえる外部状態における確実な反応でないがからである。この目的のために、『状態機械外からの状態情報の取得（Getting state information out of the machine）』において紹介されたstate_cast<>( )関数は、状態

内でも利用できる。

幾分不自然な例として、キーボード(keyboard)もまた EvRequestShutdownイベントを受け付けると仮定すると、全てのロックキーがオフ状態という状況の場合にのみ、反応がキーボードの終了を促す。Keyboard状態を次の様に変更する:

```
struct EvRequestShutdown : fsm::event< EvRequestShutdown > {};

struct NumLockOff;
struct CapsLockOff;
struct ScrollLockOff;
struct Active: fsm::simple_state<
  Active, Keyboard, fsm::custom_reaction< EvRequestShutdown >,
  mpl::list< NumLockOff, CapsLockOff, ScrollLockOff > >
{
  fsm::result react( const EvRequestShutdown & )
  {
    if ( ( state_downcast< const NumLockOff * >() != 0 ) &&
         ( state_downcast< const CapsLockOff * >() != 0 ) &&
         ( state_downcast< const ScrollLockOff * >() != 0 ) )
    {
      return terminate();
    }
    else
    {
      return discard_event();
    }
  }
};
```

参照型でなくポインタ型が渡されるならば、キャストが失敗した場合は std::bad_castが送出される代わり、0ポインタが返される。state_cast でなく state_downcastを利用する場合は注意すること。 boost::polymorphic_downcastとdynamic_castの相違に類似しているので、state_downcastはstate_castの類型よりも素早く、より多くの派生型を渡す場合に利用できる。state_castは、追加した規定クラスを問い合せたい場合に使われる。

カスタム状態の問い合せ(Custom state queries)

状態機械では、現時点に留まっている状態を正確に探し出すことが、しばしば望まれる。これはある程度、state_cast<>()やstate_downcast<>()により既に可能ではあるが、これらのユーティリティは、両者供に『あなたは状態Xですか?』という問いに対して単にイエス・ノーを答えるだけという、どちらかと言うと、制限がある。state_cast<>()に対して、状態クラスよりはむしろ追加した規定クラスを渡す場合の方が洗練された問いを尋ねることができるが、これにはより多くの作業を含み(全ての状態は追加した規定クラスから派生させる上、実装する必要がある)、遅く(state_cast<>()内でdynamic_castを使うより)、プロジェクトへコンパイル時のスイッチを「C++ RTTI」を有効にする変更を強要し、さらには状態における入退場時の速度に悪い影響を与える。

『あなたはどんな状態ですか?』という問いは、とりわけデバッグに対してより有用である。この目的に対して、state_machine::state_begin()とstate_machine::state_end()により、全ての現在の**最も深い内部**状態を繰り返し訪問することが出来る。戻されたイテレータへの参照は、全ての状態の共通の基底クラスであり、次の様にして現在の状態に関する構成内容を印刷出来る。(完全なコードに関してはKeyboard例題プログラムを参照のこと)

```
void DisplayStateConfiguration( const Keyboard & kbd )
{
  char region = 'a';

  for (
    Keyboard::state_iterator pLeafState = kbd.state_begin();
    pLeafState != kbd.state_end(); ++pLeafState )
  {
    std::cout << "Orthogonal region " << region << ": ";
    std::cout << typeid( *pLeafState ).name() << "\n";
    ++region;
  }
}
```

もし必要なら、外側の状態は、const state_machine::state_base_type のポインタを返す state_machine::state_base_type::outer_state_ptr()を利用してアクセス出来る。最も外側の状態において呼ばれた場合、この関数は単にOポインタを返す。

状態の型情報(State type information)

アプリケーションの実行時サイズを減らすためにはコンパイルスイッチの「C++ RTTI」を無効にしなければならない。これは、次の二つの関数が無いことになり、全ての現在の状態を繰り返し訪問することが役に立たなくなる:

- *unspecified_type* simple_state::static_type()
- *unspecified_type*
  state_machine::state_base_type::dynamic_type() const

両者が返す値は、operator==()とstd::less<>により比較できる。これだけでも typeid の助け無しで上記のDisplayStateConfiguration()関数の実装には十分であるが、状態名を型の情報値として使用されなければならないmapのようなものに関しては、まだ少し扱いにくい。

That's why the following functions are also provided (only available when BOOST_FSM_USE_NATIVE_RTTI is **not** defined):

そんな訳で次の関数が提供される(ただし、BOOST_FSM_USE_NATIVE_RTTI が宣言**されない**場合に有効):

- template< class T >
  void simple_state::custom_static_type_ptr(const T *);
- template< class T >
  const T * simple_state::custom_static_type_ptr();
- template< class T >
  const T * state_machine::
    state_base_type::custom_dynamic_type_ptr() const;

これらは、各々の状態に関して任意の状態型情報と直接関係づけることを許す ...

```
// ...

int main()
{
```

```
    NumLockOn::custom_static_type_ptr( "NumLockOn" );
    NumLockOff::custom_static_type_ptr( "NumLockOff" );
    CapsLockOn::custom_static_type_ptr( "CapsLockOn" );
    CapsLockOff::custom_static_type_ptr( "CapsLockOff" );
    ScrollLockOn::custom_static_type_ptr( "ScrollLockOn" );
    ScrollLockOff::custom_static_type_ptr( "ScrollLockOff" );

    // ...
  }
```

... そして次のように表示関数を書き換える:

```
  void DisplayStateConfiguration( const Keyboard & kbd )
  {
    char region = 'a';

    for (
      Keyboard::state_iterator pLeafState = kbd.state_begin();
      pLeafState != kbd.state_end(); ++pLeafState )
    {
      std::cout << "Orthogonal region " << region << ": ";
      std::cout <<
        pLeafState->custom_dynamic_type_ptr< char >() << "\n";
      ++region;
    }
  }
```

例外処理(Exception handling)

例外は、状態からの退場動作を除いた全ての利用者コードから伝播させることが出来る(廃棄にマップされた動作と廃棄処理は実質的に決してC++では例外を送出しない)。箱の外で、state_machineは以下の処理を行なう:

1. 例外はキャッチされる。
2. 補足ブロックにおいて、fsm::exception_thrownイベントはスタック上に割り当てられる。
3. 補足ブロックではまた、fsm::exception_thrownイベントは**直ち**にディスパッチが試みられる。例外が成功裏に取り扱われた後でキュー内に残ったイベントは出来る限りディスパッチされる。
4. もし例外が成功裏に取り扱われたならば、状態機械はクライアントへ正常に戻る。もし例外が成功裏に取り扱われなかったならば、原因となった例外は、例外の処理可能な状態機械のクライアントへ再送出される。

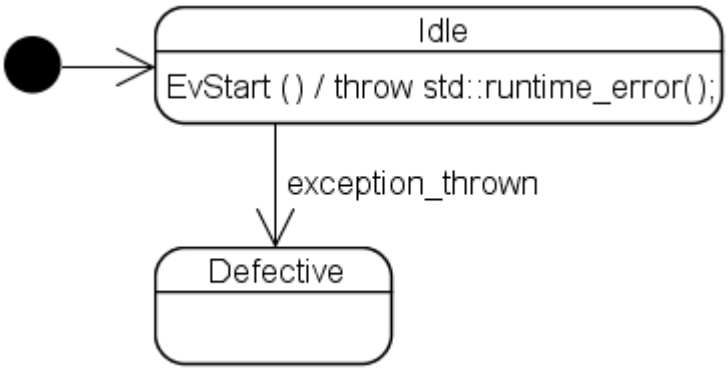
この振る舞いはexception_translatorクラスで実装されおり、そしてそれはstate_machineクラステンプレートにおけるExceptionTranslatorパラメータのデフォルトである。これは、或るプラットフォームにおいて、兎角問題のある例外処理の実装を変更したいというユーザの要望により、取り入れられた(『区別できる例外(Discriminating exceptions)』を参照のこと)。

boost::fsm はまた、例外処理を不可にしたC++によってコンパイルされたアプリケーションでも利用可能であるが、それは**全ての**エラー処理に関するサポートを失うことを意味し、扱いにくいエラー処理をより適切に作ることになる(『理論的裏付け(Rationale)』における『エラー処理(Error handling)』の項目を参照のこと)。

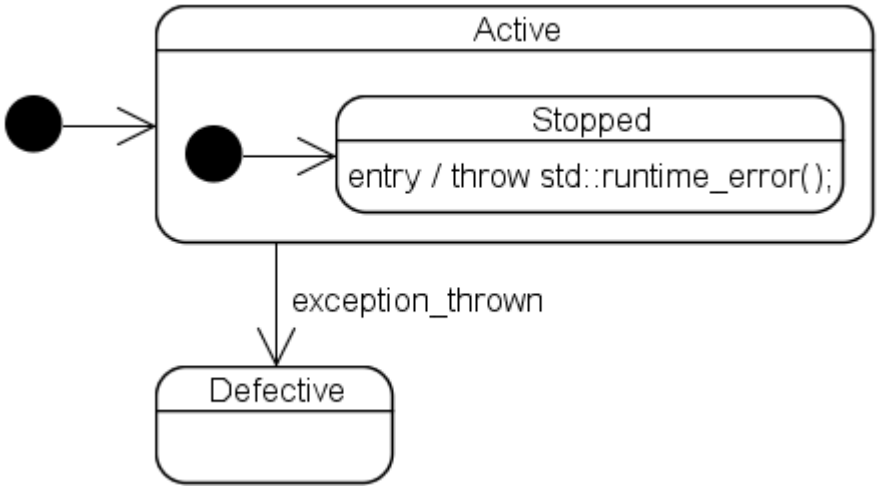
如何にして状態はfsm::exception_thrownイベントに反応するか(Which states can react to an fsm::exception_thrown event)?

これは例外が発生した場所により異なる。三通りのシナリオがある:

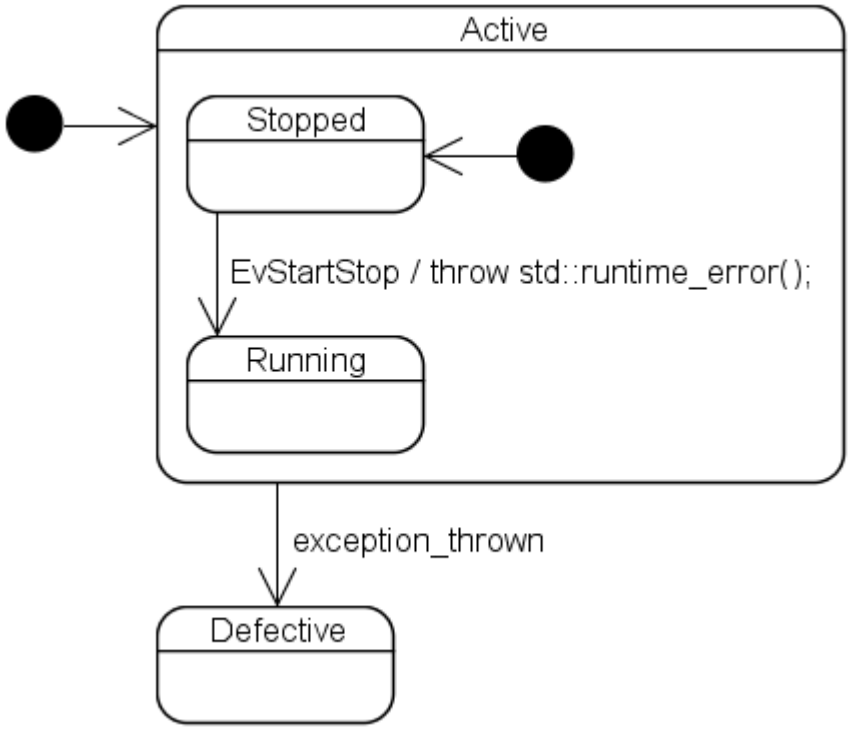
1. `react` メンバー関数は、何れかの反応関数を呼び出す**前に**例外を伝播する。例外を引き起こした状態は最初に反応を得ようとし、次の状態機械はEvStart イベントを受信した後、Defective 状態へ遷移する:



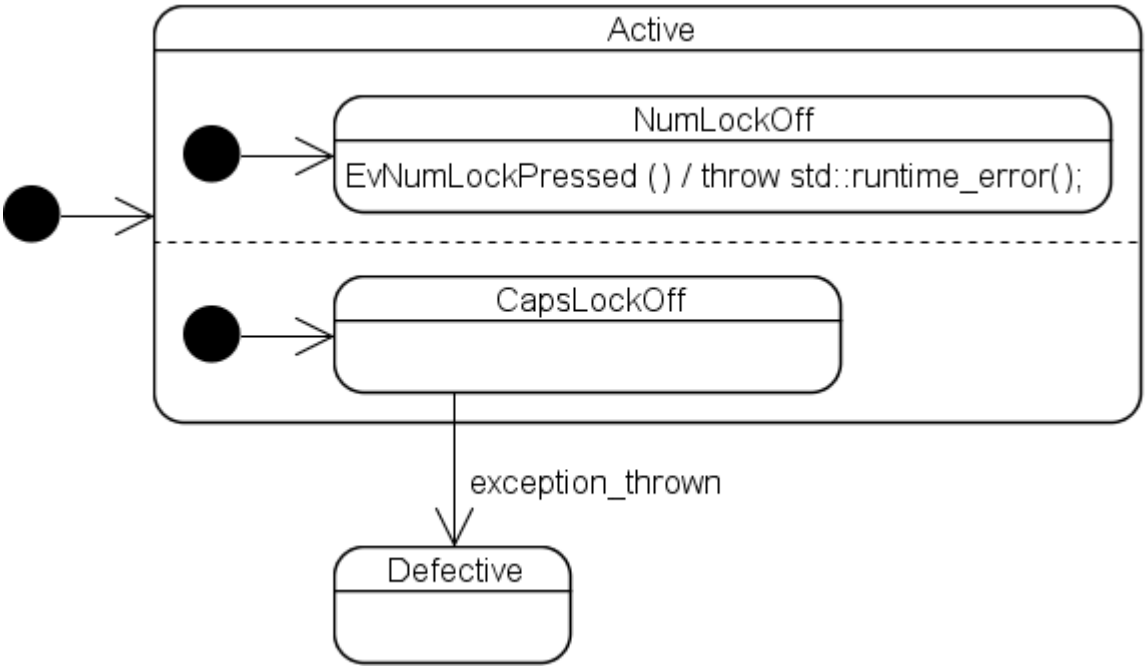
2. 状態の入場動作(コンストラクタ)は例外を伝播する。例外を起こした状態の外側の状態は最初に反応を得ようとし、次の状態機械はStopped状態へ入ろうとした後、Defective 状態へ遷移する:



3. 遷移動作は例外を伝播する。最も深い部分にある遷移元状態と遷移先状態に共通な外部状態は、最初に反応を得ようとするため、次の状態機械はEvStartStopイベントを受信した後、Defective 状態へ遷移する:



通常のイベントに関して、もし最初に試行する状態がそれを提供しない場合（または反応が明示的に`forward_event();`関数で戻った場合）、ディスパッチアルゴリズムは外側に向かって反応を探すように移動する。しかしながら、**通常のイベントと対照的に、最も外側の状態までの試行に一度失敗すると、放棄する**ため、次の状態機械はEvNumLockPressed イベントを受け取った後、Defective状態へ遷移**しない**:



その代わり、状態機械は終結させられ、元となった例外が送出される。

成功する例外処理（Successful exception handling）

例外処理は成功裏に取り扱うために良く考えられた、例えば:

- `fsm::exception_thrown`イベントの検出のための適切な反応と

- 状態機械は、その反応の完了後、安定状態となる。

二番目の要件は、この最後の節におけるシナリオ2と3のために重要である。これらのシナリオでは、例外が取り扱われた時、状態機械は遷移の途中にある。他に何もせずイベントを捨てるだけなら、状態機械は無効な状態に取り残される。

不成功な例外の取り扱いのための箱の外の振る舞いは、元となった例外を再送出することである。状態機械は、呼び出し元の状態機械へ例外を伝播する前に、終結させられる。

例外の識別（Discriminating exceptions）

fsm::exception_thrownオブジェクトはキャッチブロック内でディスパッチできるため、再送出またはカスタム反応で補足できる:

```
struct Defective : fsm::simple_state<
  Defective, Purifier > {};

// これは、清浄器状態機械における深い入れ子を模倣している
struct Idle : fsm::simple_state< Idle, Purifier,
  mpl::list<
    fsm::custom_reaction< EvStart >,
    fsm::custom_reaction< fsm::exception_thrown > > >
{
  fsm::result react( const EvStart & )
  {
    throw std::runtime_error( "" );
  }

  fsm::result react( const fsm::exception_thrown & )
  {
    try
    {
      throw;
    }
    catch ( const std::runtime_error & )
    {
      // 単にstd::runtime_errorsはDefective状態への遷移をもたらす ...
      return transit< Defective >();
    }
    catch ( ... )
    {
      // ... 全ての他の例外は外側の状態へ転送される。
     // 状態機械は終了させられ、そして例外は再送出される、
     // もし外側の状態がどちらも処理しなければ ...
      return forward_event();
    }

    // あるいは、もし直ちに状態機械を終了したければ、
   // 再送出は異なる例外を送出することもできる。
  }
};
```

不運にも、この表現方法（catchブロックが入れ子となるtry ブロック内でthrow;を利用）は、少なくと

も、非常に一般的なコンパイラのひとつで動作しない。もし、このようなプラットフォームの一つを使わなければならい場合、state_machineクラステンプレートに対してカスタマイズされた例外通知クラスを渡すことが出来る。これは、例外の型によって決まる異なったイベントを生成することを認める。

副状態機械とパラメータ化された状態(Submachines & parameterized states)

副状態機械はイベント駆動型のプログラムであり、関数はプロシージャ型のプログラムとして、しばしば必要とされる機能を実装した再利用可能な組み立てブロックとなっている。UMLに関連した表記は完全に明確ではない。それは、厳しく制限されている(例えば、同じ副状態機械は異なった直交領域として表せない)ように見えるし、例えばパラメータの様な、明確な要素について説明しているようにも見えない。

boost::fsm is completely unaware of submachines but they can be implemented quite nicely with templates. Here, a submachine is used to improve the copy-paste implementation of the keyboard machine discussed under

boost::fsm は副状態機械を完全には認知していないが、テンプレートを使い非常にうまく実装できる。ここで、副状態機械は、『直交状態(Orthogonal states)』節において議論されたキーボード状態機械のコピペ実装を改善するために使用される: Orthogonal states:

```
enum LockType
{
  NUM_LOCK,
  CAPS_LOCK,
  SCROLL_LOCK
};

template< LockType lockType >
struct Off;
struct Active : fsm::simple_state<
  Active, Keyboard, fsm::no_reactions, mpl::list<
  Off< NUM_LOCK >, Off< CAPS_LOCK >, Off< SCROLL_LOCK > > > {};

template< LockType lockType >
struct EvPressed : fsm::event< EvPressed< lockType > > {};

template< LockType lockType >
struct On : fsm::simple_state<
  On< lockType >, Active::orthogonal< lockType >,
  fsm::transition< EvPressed< lockType >, Off< lockType > > > {};

template< LockType lockType >
struct Off : fsm::simple_state<
  Off< lockType >, Active::orthogonal< lockType >,
  fsm::transition< EvPressed< lockType >, On< lockType > > > {};
```

非同期状態機械(Asynchronous state machines)

非同期状態機械の必要性(Why asynchronous state machines are necessary)

名前が連想させるように、同期状態機械は同時に発生した各々のイベントを同期させて処理する。この振る舞いはstate_machine<>クラステンプレートによって実装されており、そのprocess_event()は全ての反応が実行された後に限って戻ることができる(内部イベントによって引き起こされたかもしれない動作も含む)。そのうえ、この関数はまた厳格に再入が不可である(全ての他の関数と同じように、

state_machine<>もスレッドセーフではない)。これを作るのは、二つのstate_machine<>副クラスにとって、双方向形式によるイベントを介して正確に通信することは難しく、**シングルスレッドのプログラムではなおのこと**である。例えば、状態機械Aは外側からのイベントを処理中とする。動作内部では、状態機械Bに(適切なイベントと伴にB::process_eventを呼ぶことにより)新しいイベントを送信することを決定する。このとき、boost::functionに似たコールバックを介して、Bからの応答が戻るまで"待つ"が、その為のA::process_eventの参照をイベントのデータメンバーとして渡す。しかしながら、AがBからのイベントの送信応答を"待って"いる間、A::process_eventが外部イベント処理からまだ戻らないか、コールバックを介してBが応答した直後、A::process_eventは**不可避的**に再入される。これは、シングルスレッドでは全て現実的に起こるため、"待ち"に引用符を付けている。

どのように作動するか(How it works)

state_machine<>とは対照的に、 asynchronous_state_machine<>にはメンバー関数 process_event() がない。その代わりにqueue_event()だけがあり、それはイベントをキューに登録した後、直ちに戻る。 workerスレッドは、それを処理した後でキューからイベントを取り出す。アプリケーションがboost::thread ライブラリを使う理由は、worker<>クラスにおいて、鍵を掛けたり外したりする必要があることと待ち処理の利用が容易であることによる。

アプリケーションは通常、最初にworker<>オブジェクトを作成し、それからworkerオブジェクトのコンストラクタを通して、ひとつまたはそれ以上の asynchronous_state_machine<>サブクラスオブジェクトを作成する。最後に、worker<>::operator()() 関数は、現スレッドにおいて実行を許可された状態機械か、新しくboost::threadへ渡されたboost::functionオブジェクトが参照しているoperator()関数のどちらかにより直接呼ばれる。次のコードでは、一方の状態機械を新しいboost::thread内で、もう片方はメインスレッド内で走らせている(完全なソースコードはPingPong例題プログラムを参照のこと):

```
// ...

struct Waiting;
struct Player :
  fsm::asynchronous_state_machine< Player, Waiting >
{
  typedef fsm::asynchronous_state_machine< Player, Waiting >
    BaseType;

  Player( fsm::worker<> & myWorker ) :
    BaseType( myWorker ) // ...
  {
    // ...
  }

  // ...
};

// ...

int main()
{
  fsm::worker<> worker1;
  fsm::worker<> worker2;

  // 各プレイヤーは自身のworkerスレッドで実行する
  Player player1( worker1 );
```

```
  Player player2( worker2 );

  // ...

  // 新しいスレッド内の最初のworkerを実行
  boost::thread otherThread(
    boost::bind( &fsm::worker<>::operator(), &worker1 ) );

  worker2(); // このスレッド内の二番目のworkerを実行
  otherThread.join();

  return 0;
}
```

ふたつのboost::threadsを同様に使う:

```
int main()
{
  // ...

  boost::thread thread1(
    boost::bind( &fsm::worker<>::operator(), &worker1 ) );
  boost::thread thread2(
    boost::bind( &fsm::worker<>::operator(), &worker2 ) );

  // do something else ...

  thread1.join();
  thread2.join();

  return 0;
}
```

または、同じworkerスレッドにおいて、両方の状態機械を走らせる:

```
int main()
{
  fsm::worker<> worker1;

  Player player1( worker1 );
  Player player2( worker1 );

  // ...

  worker1();

  return 0;
}
```

worker<>::operator()() 関数は、先ず全ての状態機械を初期化し、その後イベント待ちとなる。queue_event関数が以前登録された状態機械のひとつから呼ばれる場合はいつも、渡されたイベントはworkerのキューに登録され、そしてそのworkerスレッドは再び待ち状態となる前に、キューされた全てのイベントをディスパッチするために起こされる。worker<>::operator()()関数は、全ての状態機

械が終結すると直ちに戻る。worker<>::operator()()はまた、状態機械が処理に失敗した場合の例外を送出する。この場合、全ての状態機械は、例外が伝播する前に終了させられる。

注意事項:

- **asynchronous_state_machine<>サブクラスオブジェクトは、worker::operator()()関数が戻る前に破棄されなければならない。さらに、worker<>オブジェクトは、登録された全ての状態機械が廃棄された後でのみ、破壊可能である。これらの規則に違反することは実行時に表明エラーという結果を招くことになる。**
- **asynchronous_state_machineのインターフェースは、コンストラクタとqueue_event()関数だけで構成される。技術的な理由により、initiate()、process_event()などの他の関数を公に利用できるとは言っても、worker以外のいかなるスレッドから、それらの関数を呼び出すことは安全でない(難しいため大抵のユーザーは管理できない)。**
  asynchronous_state_machine<>::queue_event() 関数は、複数のスレッドから同時に安全に呼び出せる唯一の関数である。

Revised 18 November, 2003