Настоящий документ описывает подход к построению монитора событийного моделирования, который будет использован в проекте «NM Model».

1 Задачи нового монитора

Прежде всего, новый монитор нужен для эффективного моделирования прерываний. Сложность при этом в том, что прерывание может прийти в произвольный момент времени. Для точного моделирования в рамках существующего монитора необходима синхронизация моделей на каждой инструкции (то есть по одному событию в каждой модели на инструкцию), что мало приемлемо. Поэтому предполагается «оптимистическая схема» -- модель процессора может просчитать данные вперед на какое-то количество инструкций. В случае если прерывания не происходит, моделирование продолжается дальше. Иначе, модель процессора «откатывается» к моменту возникновения прерывания. От монитора при этом требуется:

- Механизм откатов. С точки зрения монитора, это означает что модель может сказать «Следующее событие будет в момент времени 10, если мне не прийдет прерывание.». При возникновении прерывания модель может поменять время следующего события
- Максимальной эффективности при обработке событий.

Кроме этого, монитор должен быть достаточно гибким и в частности поддерживать:

- Динамическую конфигурацию (за пределами «создать N экземпляров процесса X»). Пользователь должен иметь возможность добавить модель нового устройства почти что методом «drag-and-drop». Например, добавление чего-либо в модель на MM-языке, особенно иерархическую, крайне неудобно.
- Реализацию моделей устройств в виде реактивных моделей (обработчкиков событий) или в виде моделей с отдельным потоком управления.

2 Проблемы имеющихся мониторов

Сейчас есть три монитора – libmon (Dyana), runmodel (Nikita) и unnamed (Suhoi). Последний, по словам Никиты, для повторного использования не очень подходит. Остаются два. Общая проблема – не очень хорошее качество кода и монолитная структура.

2.1 libmon

Все (Никита, Макс, я) считают libmon написанным через пень-колоду и неподдерживаемым. Все модели исполняются в отдельных потоках, это свойство глубоко вшито во всю реализацию и не может быть изменено.

Возможность отката событий не предусмотрена и нет очевидной зацепки, чтобы убрать текущее событие из календаря.

Код сильно завязан на MM-язык и код, который создает mmt.

2.2 runmodel

По сути, не поддерживается и не использовался на практике.

Структура реализации не оптимальна – огромный класс реализующий все возможности. Класс – singleton, вся информация о процессе моделирования в статических полях. Документация неполная (или хорошо спрятана). Возможность отката событий есть. По умолчанию все модели выполняются в одном потоке с чередованием. Есть возможность создать отдельных поток для модели, как именно пользоваться непонятно.

3 Проект нового монитора

Основная проблема существующих решений – монолитный дизайн, в который непонятно как встраиваться. Чтобы с этим бороться, предлагается разбить новый монитор по уровням:

- Event loop. Уровень, решающий какая из моделей должна выполняться следующей. На этом уровне нет ни очередей событий, ни даже событий – лишь набор моделей и время следующего события для каждой.
- Adaptors. Уровень, предоставляющие более удобные средства написания моделей:
  - Event_model – модель с локальной очередью событий
  - Threaded_model – wrapper, исполняющий любую модель в отдельном потоке (до конца еще не придуман)
- Structure. Уровень, предоставляющий специфичные для каждого проекта структурные элементы. Например, для «Модели NM» на этом уровне будут находится классы для конкретных видов моделируемых плат.

3.1 Уровень Event loop

Уровень состоит из двух сущностей – моделей (классо Simulation_model) и драйвера (класс Simulator). Драйвер запускает по очереди все модели. Каждая модель отрабатывает внутренную логику до того момента, когда происходит какое-либо гловальное событие. Время этого события возвращается драйверу. Определив минимальное время глобального события, драйвер устанавливает текущее модельное время равным этому минимальному времени, и запускает модель, заявившую это минимальное время. Модель выполняет событие и продолжает моделирование до следующего глобального события.

В процессе обработки события модель может взаимодействовать с другими моделями. Например, при обработке события «посылка сообщения» возможно планирование события «прием сообщения» в другой модели. При таком взаимодействии другие модели могут решить, что ранее заявленное время возникновения события неверно, и изменить его как в сторону увеличения, так и в сторону уменьшения (вплоть до времени текущего события).

Интерфейсы

```
class Simulation_model {
public:
      /** Создает новую модель и добавляет ее к списку моделей, которыми
          управлениет owner. */
      Simulation_model(Simulator* owner);

      /** Выполняет текущее запланированние событие и моделирует дальшейшее
          исполнение, до следующего события. Возвращает время следующего
          события в попугаях.
          Возвращаемое значения -1 говорит о завершении модели.
          Метод вызывается драйвером.
      */
      virtual unsigned long long simulate() = 0;

      /** Аналогично simulate, но никогда не моделирует дальше
          переданного времени.
      */
      virtual unsigned long long simulate_until(unsigned long long max) = 0;

      /** Метод вызывается унаследованными классами, и сообщает что
```

```
            время следующего события изменилось и равно 'time'.
        void backtrace(unsigned long long time);
};


class Simulator
{
public:
     /* Пытается выполнить следующее событие. Возвращает true если
        удалось, и false если нет (все модели завершили выполнение).
     */
     bool execute_next_event();

     /* Может быть вызван только из метода simulate одной из моделей.
        Возвращает модель, которая выполняется с настоящий момент.
        Предназначен, чтобы избежать протакскивания дополнительных параметров
        во все функции кода, что может быть неудобно.
     */
     Simulation_model* current_model() const;

     /* Задает максимально модельное время, которое будет промоделировано. */
     void set_max_time(unsigned long long time);

     /* Hook, вызываемый при продвижении модельного времени. Возможно,
        вызывается не при каждом продвижении и может быть использован только
        для интерфейса с пользователем.
     */
     boost::function<void ()(unsigned long long)> model_time_changed;
}
```

Эти интерфейсы не предполагают явный класс «событие», и не требуют, чтобы взаимодействие между моделями шло через драйвер (за исключением синхронизации времени). В значительной части случаем взаимодействие будет делаться прямым вызовом методов других моделей. Это позволяет сделать драйвер независимым от конкретных моделей – ему нет необходимости знать о всех типах событий которые только могут быть.

3.2 Уровень Adaptors

Класс Event_model предназначен для создания моделей с локальной очередью сообщений. Например, процессору может быть почти одновременно послано несколько прерываний и все их нужно где-то хранить.

Интерфейс класса:

```
class Event_model : public Simulation_model {
public: // Simulation_model overrides
      /** If local event queue is empty, call 'simulate_until' with
          infinity as parameter and returns its return value.
          Otherwise, calls 'simulate_until' with the minimum time of
          event in local queue. Returns that min time.
      */
      unsigned long long simulate();

protected:
      struct Event
      {
          unsigned long long time;
          boost::function<void ()> handler;
      }
      void queue_event(const Event&);
};
```

3.3 Пример: обработка прерываний

Три модели процессора последовательно запускаются. Первые две модели полностью просчитывают 100 тактов работы процессора. Третья модель на 20 такте обрабатывается код посылки прерывания в первый процессора. То есть, набор времен следующих событий будет [100, 100, 20]. Драйвер запускает третью модель. При этом моделируется посылка прерывания и первой модели сообщается о прерывании в момент времени 30 (10 тактов на передачу). Модель откатывает состояние процессора до 30-го такта и сообщает драйверу, что следующее событие будет не в момент времени 100, а в момент времени 30. Набор времен следующих событий – [30, 100, 120]. На выполнение запускается первая модель которая моделирует обработку пришедшего прерывания.

3.4 Пример: посылка сообщения пассивному получателю

Пассивный получатель (модель устройства) выдает в качестве времени следующего события бесконечность. Другая модель обрабатывает посылку сообщения модели устройства. Модель сообщает драйверу что время следующего события изменилось.

3.5 Пример: посылка сообщения активному получателю

Под активным получателем имеется в виду модель, которая что-то делает и в отсутствие сообщений, но при этом сообщения не прерывают работу. То есть, имеется блокирующий или неблокирующий receive.

Для неблокирующего receive, в модели вводится специальный тип события «вызов функции recieve», и время следующего такого события сообщается монитору. Когда событие обрабатывается, все ранее произошедные посылки сообщений уже отработаны, и можно установить, есть ли сообщения для текущей модели или нет.

Для блокирующего receive есть два варианта. Если нет необходимости запоминать момент времени, когда мы заблокировались на receive, то обработка блокирующего recieve сводится к возвращению бесконечно большого времени следующего события. После прихода сообщения моделирование продолжается с времени прихода сообщения. Если надо запоминать момент времени, когда мы заблокировались, вводится сообщения «вызов функции receive».