Manifold 0.9 Simulation Kernel API

February 21, 2013

1 Introduction

The Manifold simulation kernel provides a rich set of functions that allows the user to easily build a parallel discrete event simulation program. A typical Manifold simulation program includes the following sequence of steps:

1. Call `manifold::kernel::Manifold::Init()` function.
2. Create the components with the `manifold::kernel::Component::Creat()` family of functions.
3. Create clock objects.
4. Register components with clocks, if applicable.
5. Connect components with the `manifold::kernel::Manifold::Connect()` family of functions.
6. Set a simulation stop time with one of the `manifold::kernel::Manifold::StopAt()` functions.
7. Call `manifold::kernel::Manifold::Run()` to start the simulation.
8. At the end, call `manifold::kernel::Manifold::Finalize()`.

This document describes the public functions that make up the Manifold simulation kernel's API. The functions are divided into the following categories:

- Simulation functions
- Component functions
- Clock functions

A separate section describes data serialization and de-serialization.

2 Simulation Functions

This set of functions can be further divided into the following categories:

- Simulation control
- Event scheduling

- Component connection

These functions are defined in the class `manifold::kernel::Manifold`.

2.1 Simulation Control Functions

```
Init(int argc, char** argv, SchedulerType type,
SyncAlg::SyncAlgType_t syncAlg, Lookahead::LookaheadType_t la)
```

This is the initialization function and should be called at the beginning of the simulator program.

Input parameters

- `argc` and `argv`: standard C/C++ command-line parameters
- `type`: The scheduler type could be TICKED, TIMED, or MIXED.
- `syncAlg`: Synchronization algorithm. Current SA_LBTS and SA_CMB are supported.
- `la`: Look-ahead type. Current only LA_GLOBAL (global look-ahead) is allowed.

Return value

None.

```
Finalize()
```

This function should be called at the end of the siThese functions are defined in the class `manifold::kernel::Manifold`.mulation program, before the program exits.

```
StopAtTime(Time_t tm), StopAt(Ticks_t tick), StopAtClock(Ticks_t
tick, Clock& clk)
```

These functions specify when the simulation should end. `StopAtTime()` specifies the end time in seconds (simulation time). `StopAt()` specifies the end time in terms of ticks of the master clock, the very first clock created in the system. `StopAtClock()` specifies the end time in ticks of the given clock.

Input parameters

- `tm`: Stop time in seconds.
- `tick`: Stop time in ticks.
- `clk`: The given clock. These functions are defined in the class `manifold::kernel::Manifold`.

Return value

None

`Run()`

This function starts the simulation. It should be called after the initialization actions have finished and the simulation is ready to start.

`Stop()`

This function terminates the simulation main loop.

`Now(), NowTicks(), NowTicks(Clock& clk), NowHalfTicks(), NowHalfTicks(Clock& clk)`

These functions return the current simulation time in seconds, ticks of the master clock, ticks of a given clock, half ticks of the master clock, or half ticks of a given clock, respectively.

Input parameters

clk: The given clock.

Return value

The current simulation time in seconds, ticks, or half ticks.

`GetRank()`

This function returns the MPI rank of the LP.

2.2 Event Scheduling Functions

This family of functions allows user to schedule future events.

- The time when the future event occurs is relative to the current time, and can be specified in seconds, ticks, or half ticks.
- The event handler can take 0-4 input parameters.

2.2.1 Scheduling in ticks with respect to the master clock

```
template<typename T, typename OBJ>
static TickEventId Schedule(Ticks_t t, void(T::*handler)(void), OBJ*
obj),
```

```
template<typename T, typename OBJ, typename U1, typename T1>
static TickEventId Schedule(Ticks_t t, void(T::*handler)(U1), OBJ*
obj, T1 t1),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2>
static TickEventId Schedule(Ticks_t t, void(T::*handler)(U1, U2),
OBJ* obj, T1 t1, T2 t2),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3>
static TickEventId Schedule(Ticks_t t, void(T::*handler)(U1, U2, U3),
OBJ* obj, T1 t1, T2 t2, T3 t3),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3, typename U4, typename T4>
static TickEventId Schedule(Ticks_t t, void(T::*handler)(U1, U2, U3,
U4), OBJ* obj, T1 t1, T2 t2, T3 t3, T4 t4),
```

These functions schedule an event for a given object with the given member function as the event handler. The time when the event should occur is specified in ticks from now with respect to the master clock. They differ in the number of parameters that the handler takes. The number can be 0 to 4.

Input parameters

- `t`: This is when the event should occur. The time is in ticks of the master clock from the current time.
- `handler`: This is a member function that is called at the given time.
- `obj`: The pointer to the object whose member function `handler` is to be called.
- `t1,t2,t3,t4`: Input parameters to the `handler` function.

2.2.2 Scheduling in ticks with respect to a given clock

```
template<typename T, typename OBJ>
static TickEventId ScheduleClock(Ticks_t t, Clock& clk,
void(T::*handler)(void), OBJ* obj),

template<typename T, typename OBJ, typename U1, typename T1>
static TickEventId ScheduleClock(Ticks_t t, Clock& clk,
void(T::*handler)(U1), OBJ* obj, T1 t1),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2>
static TickEventId ScheduleClock(Ticks_t t, Clock& clk,
void(T::*handler)(U1, U2), OBJ* obj, T1 t1, T2 t2),

template<typename T, typename OBJ, typename U1, typename T1 typename
```

```
U2, typename T2 typename U3, typename T3>
static TickEventId ScheduleClock(Ticks_t t, Clock& clk,
void(T::*handler)(U1, U2, U3), OBJ* obj, T1 t1, T2 t2, T3 t3),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3, typename U4, typename T4>
static TickEventId ScheduleClock(Ticks_t t, Clock& clk,
void(T::*handler)(U1, U2, U3, U4), OBJ* obj, T1 t1, T2 t2, T3 t3, T4
t4),
```

These functions schedule an event for a given object with the given member function as the event handler. The time when the event should occur is specified in ticks from now with respect to a given clock. They differ in the number of parameters that the handler takes. The number can be 0 to 4.

Input parameters

- `t`: This is when the event should occur. The time is in ticks of the given clock from the current time.
- `clk`: The given clock.
- `handler`: This is a member function that is called at the given time.
- `obj`: The pointer to the object whose member function `handler` is to be called.
- `t1,t2,t3,t4`: Input parameters to the `handler` function.

2.2.3 Scheduling in half ticks with respect to the master clock

```
template<typename T, typename OBJ>
static TickEventId ScheduleHalf(Ticks_t t, void(T::*handler)(void),
OBJ* obj),

template<typename T, typename OBJ, typename U1, typename T1>
static TickEventId ScheduleHalf(Ticks_t t, void(T::*handler)(U1),
OBJ* obj, T1 t1),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2>
static TickEventId ScheduleHalf(Ticks_t t, void(T::*handler)(U1, U2),
OBJ* obj, T1 t1, T2 t2),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3>
static TickEventId ScheduleHalf(Ticks_t t, void(T::*handler)(U1, U2,
U3), OBJ* obj, T1 t1, T2 t2, T3 t3),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3, typename U4, typename T4>
static TickEventId ScheduleHalf(Ticks_t t, void(T::*handler)(U1, U2,
```

```
U3, U4), OBJ* obj, T1 t1, T2 t2, T3 t3, T4 t4),
```

These functions schedule an event for a given object with the given member function as the event handler. The time when the event should occur is specified in **half ticks** from now with respect to the master clock. They differ in the number of parameters that the handler takes. The number can be 0 to 4.

Input parameters

- `t`: This is when the event should occur. The time is in half ticks of the master clock from the current time.
- `handler`: This is a member function that is called at the given time.
- `obj`: The pointer to the object whose member function `handler` is to be called.
- `t1,t2,t3,t4`: Input parameters to the `handler` function.

2.2.4 Scheduling in half ticks with respect to a given clock

```
template<typename T, typename OBJ>
static TickEventId ScheduleClockHalf(Ticks_t t, Clock& clk,
void(T::*handler)(void), OBJ* obj),

template<typename T, typename OBJ, typename U1, typename T1>
static TickEventId ScheduleClockHalf(Ticks_t t, Clock& clk,
void(T::*handler)(U1), OBJ* obj, T1 t1),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2>
static TickEventId ScheduleClockHalf(Ticks_t t, Clock& clk,
void(T::*handler)(U1, U2), OBJ* obj, T1 t1, T2 t2),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3>
static TickEventId ScheduleClockHalf(Ticks_t t, Clock& clk,
void(T::*handler)(U1, U2, U3), OBJ* obj, T1 t1, T2 t2, T3 t3),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3, typename U4, typename T4>
static TickEventId ScheduleClockHalf(Ticks_t t, Clock& clk,
void(T::*handler)(U1, U2, U3, U4), OBJ* obj, T1 t1, T2 t2, T3 t3, T4
t4),
```

These functions schedule an event for a given object with the given member function as the event handler. The time when the event should occur is specified in **half ticks** from now with respect to a given clock. They differ in the number of parameters that the handler takes. The number can be 0 to 4.

Input parameters

- `t`: This is when the event should occur. The time is in half ticks of the given clock from the current time.
- `clk`: The given clock.
- `handler`: This is a member function that is called at the given time.
- `obj`: The pointer to the object whose member function `handler` is to be called.
- `t1,t2,t3,t4`: Input parameters to the `handler` function.

2.2.5 Scheduling in seconds

```
template<typename T, typename OBJ>
static TickEventId ScheduleTime(double t, void(T::*handler)(void),
OBJ* obj),

template<typename T, typename OBJ, typename U1, typename T1>
static TickEventId ScheduleTime(double t, void(T::*handler)(U1), OBJ*
obj, T1 t1),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2>
static TickEventId ScheduleTime(double t, void(T::*handler)(U1, U2),
OBJ* obj, T1 t1, T2 t2),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3>
static TickEventId ScheduleTime(double t, void(T::*handler)(U1, U2,
U3), OBJ* obj, T1 t1, T2 t2, T3 t3),

template<typename T, typename OBJ, typename U1, typename T1 typename
U2, typename T2 typename U3, typename T3, typename U4, typename T4>
static TickEventId ScheduleTime(double t, void(T::*handler)(U1, U2,
U3, U4), OBJ* obj, T1 t1, T2 t2, T3 t3, T4 t4),
```

These functions schedule an event for a given object with the given member function as the event handler. The time when the event should occur is specified in seconds from now. They differ in the number of parameters that the handler takes. The number can be 0 to 4.

Input parameters

- `t`: This is when the event should occur. The time is in seconds from the current time.
- `handler`: This is a member function that is called at the given time.
- `obj`: The pointer to the object whose member function `handler` is to be called.
- `t1,t2,t3,t4`: Input parameters to the `handler` function.

2.3 Component Connection Functions

This family of functions are used two components: the source component's output is connected to the destination component's input. When calling these functions, the user must specify an event handler, which should be a member function of the destination component's class. Also required is the link delay, which can be specified in seconds, ticks, or half ticks.

```
template<typename T, typename T2>
static void Connect(CompId_t src, int srcIdx, CompId_t dst, int
dstIdx, void (T::*handler)(int, T2), Ticks_t latency) throw
(LinkTypeMismatchException)
```

This function connects a source component's output port to a destination component's input port. The link delay specified is in ticks with respect to the master clock.

Input parameters

- `src`: The component ID of the source component.
- `srcIdx`: The index of the particular output port of the source component. The port index starts with 0.
- `dst`: The component ID of the destination component.
- `dstIdx`: The index of the particular input port of the destination component. The port index starts with 0.
- `handler`: This is a member function of the destination component's class. The function takes two parameters. The first is an integer that specifies the input port. The second is the data being sent over the link.
- `latency`: The link delay in ticks with respect to the master clock.
- `LinktypeMismatchException`: When a source component's output port has multiple fanouts, the event handlers from each of the fanout must expect the same data type from the source. This exception is thrown when a handler is given that expects the incorrect data type.

```
template<typename T, typename T2>
static void ConnectClock(CompId_t src, int srcIdx, CompId_t dst, int
dstIdx, Clock& clk, void (T::*handler)(int, T2), Ticks_t latency)
throw (LinkTypeMismatchException)
```

This function connects a source component's output port to a destination component's input port. The link delay specified is in ticks with respect to a given clock.

Input parameters

- `src`: The component ID of the source component.
- `srcIdx`: The index of the particular output port of the source component. The port index starts with 0.
- `dst`: The component ID of the destination component.
- `dstIdx`: The index of the particular input port of the destination component. The port index starts with 0.
- `clk`: The given clock.
- `handler`: This is a member function of the destination component's class. The function takes two parameters. The first is an integer that specifies the input port. The second is the data being sent over the link.
- `latency`: The link delay in ticks with respect to the given clock.
- `LinktypeMismatchException`: When a source component's output port has multiple fanouts, the event handlers from each of the fanout must expect the same data type from the source. This exception is thrown when a handler is given that expects the incorrect data type.

```
template<typename T, typename T2>
static void ConnectHalf(CompId_t src, int srcIdx, CompId_t dst, int
dstIdx, void (T::*handler)(int, T2), Ticks_t latency) throw
(LinkTypeMismatchException)
```

This function is the same as `Connect()` except the latency specified is in half ticks of the master clock.

```
template<typename T, typename T2>
static void ConnectClockHalf(CompId_t src, int srcIdx, CompId_t dst,
int dstIdx, Clock& clk, void (T::*handler)(int, T2), Ticks_t latency)
throw (LinkTypeMismatchException)
```

This function is the same as `ConnectClock()` except the latency specified is in half ticks of the given clock.

```
template<typename T, typename T2>
static void ConnectTime(CompId_t src, int srcIdx, CompId_t dst, int
dstIdx, void (T::*handler)(int, T2), Time_t latency) throw
(LinkTypeMismatchException)
```

This function connects a source component's output port to a destination component's input port. The link delay specified is in seconds.

Input parameters

- `src`: The component ID of the source component.
- `srcIdx`: The index of the particular output port of the source component. The port index starts with 0.
- `dst`: The component ID of the destination component.
- `dstIdx`: The index of the particular input port of the destination component. The port index starts with 0.
- `handler`: This is a member function of the destination component's class. The function takes two parameters. The first is an integer that specifies the input port. The second is the data being sent over the link.
- `latency`: The link delay in seconds.
- `LinktypeMismatchException`: When a source component's output port has multiple fanouts, the event handlers from each of the fanout must expect the same data type from the source. This exception is thrown when a handler is given that expects the incorrect data type.

3 Component Functions

A Manifold component should be a subclass of the class `manifold::kernel::Component`. This class provides the following general component related functions:

```
template<typename T>
static CompId_t Create(LpId_t lp, CompName_t name)

template<typename T, typename T1>
static CompId_t Create(LpId_t lp, T1& t1, CompName_t name)

template<typename T, typename T1>
static CompId_t Create(LpId_t lp, const T1& t1, CompName_t name)

template<typename T, typename T1, typename T2>
static CompId_t Create(LpId_t lp, T1& t1, T2& t2, CompName_t name)

template<typename T, typename T1, typename T2>
static CompId_t Create(LpId_t lp, const T1& t1, const T2& t2,
CompName_t name)

template<typename T, typename T1, typename T2, typename T3>
static CompId_t Create(LpId_t lp, const T1& t1, const T2& t2, const
T3& t3, CompName_t name)

template<typename T, typename T1, typename T2, typename T3, typename
T4>
```

```
static CompId_t Create(LpId_t lp, const T1& t1, const T2& t2, const
T3& t3, const T4& t4, CompName_t name)
```

These functions are called to create a component. They differ in the number of parameters the component's constructor takes. The constructor can take up to 4 parameters. Note that the user should not call the constructor directly to create a component. Instead, one of the Create() functions should be used.

Input parameters

- `lp`: The ID of the logical process to which the component is assigned.
- `t1,t2,t3,t4`: Input parameters to the constructor.
- `name`: This is an optional parameter. The default is “none”.

Return value

These functions return the integer component ID.

```
static Component* Getcomponent(CompId_t id)
```

This function returns a pointer to a component object given its component ID.

Input parameters

- `id`: The component's ID.

Return value

A pointer to the component object. It returns 0 if the component is assigned to a different LP.

```
template<typename T>
void Send(int port, T data)

template<typename T>
void SendTick(int port, T data, Ticks_t tick)

template<typename T>
void SendTime(int port, T data, Time_t tm)
```

These functions are called by a component to send its output. They differ in the delay that is used for scheduling an event for the receiver. `Send()` uses the delay specified when the components are connected. `SendTick()` and `SendTime()` use the given value as the delay, in ticks and seconds respectively, overriding the value specified when the components are connected. `SendTick()` should only be used if the link delay was specified in ticks. And `SendTime()` should only be used if the link delay was specified in seconds.

Input parameters

- `port`: The index of the output port.
- `data`: The data to be sent. It can be an object or a pointer. Reference is not allowed. If it is a pointer, the sender should not de-allocate the memory.
- `tick`: The delay to be used, in ticks.
- `tm`: The dealy to be used, in seconds.

Return vlaue

None

4 Clock Functions

These functions are defined in the class `manifold::kernel::Clock`.

```
static Clock& Master()
```

Returns the master clock, the first clock created in the system.

```
Now()
```

Returns the current tick of the master clock.

```
Now(Clock& clk)
```

Returns the current tick of the given clock.

```
NowHalf()
```

Returns the current half tick of the master clock.

```
NowHalf(Clock& clk)
```

Returns the current half tick of the given clock.

```
template<typename O>
static tickObjBase* Register(O* obj, void(O::*rising)(void),
void(O::*falling)(void))
template<typename O>
static tickObjBase* Register(Clock& clk, O* obj, void(O::*rising)
(void), void(O::*falling)(void))
```

These functions allow a component to register up to two functions with the master clock or the given

clock. The two functions are respectively called at the rising and falling edges or the clock each cycle.

Input parameters

- `obj`: The component object.
- `rising, falling`: Member functions of obj. The member functions accept no inputs and returns no values.

```
template<typename O>
static void Unregister(Clock& clk, O* obj)
```

This function unregisters a component.

Input parameters

- `clk`: The clock from which to unregister.
- `obj`: The component object.

5 Data Serialization and De-serialization

When a component sends data to another, if the two components are assigned to different LPs, then the data must be serialized at the sending LP and de-serialized at the receiving LP. Manifold provides default serialization and de-serialization functions. In most cases, the default functions should suffice.

However, the user has to write customized serialization / de-serialization functions if the data type contains one or more of the following:

- pointers
- references
- STL containers, including strings
- an object that contains one or more of the above

Writing customized serialization / de-serialization functions essentially requires writing three template specialization functions in the `manifold::kernel namespace`, as shown in the following example.

```
namespace manifold {
namespace kernel {

template<>
size_t Get_serialize_size<MyDataType>(const MyDataType* data);
```

```
template<>
size_t Serialize<MyDataType>(MyDataType* data, unsigned char* buf);

template<>
MyDataType* Deserialize<MyDatatype>(unsigned char* buf);

} //namespace kernel
} //namespace manifold
```

```
template<>
size_t Get_serialize_size<MyDataType>(const MyDataType* data)
```

This function returns the size in bytes that is required for the serialized data.

```
template<>
size_t Serialize<MyDataType>(MyDataType* data, unsigned char* buf);
```

This function serialize the data into a stream of bytes and store the results in a buffer.

```
template<>
MyDataType* Deserialize<MyDatatype>(unsigned char* buf);
```

This function takes a byte stream stored in a buffer and converts the bytes into an object of the given type.