Introduction to tempus

Certain features of tempus are Patent Pending
Contact MZA for details of our proprietary claims
Computer simulation has become an important tool in many fields of endeavor, from science and engineering to computer based training and computer animation. Over the years considerable progress has been made in tools and methodologies for simulation, but much of this progress has come in the form of improvements to a variety of relatively specialized tools, for modeling control systems, flexible structures, fluid dynamics, communication networks, and so forth. By comparison, relatively little progress had been made in tools designed to support interdisciplinary simulation, involving interactions among subsystems with qualitatively dissimilar behaviors and requiring differing modeling approaches.

tempus is a simulation executive that uses a powerful and flexible block diagram-based architecture designed to meet the demands of interdisciplinary simulation. Combining ideas from object-oriented programming and hybrid simulation, tempus can be used to model just about anything. It has an open architecture, which makes it easy to integrate other software into tempus, and vice versa. This course provides an introduction to the application of tempus to the development of large, complex, and interdisciplinary models.
Course Objectives

- Explain the motivation for the existence and design of tempus.
- Explain how to use tempus.
- Explain how to develop models with tempus, including developing new source code capabilities.

The terminologies of computer programming and simulation are not always standardized. The terms and concepts used throughout this lesson may have broader meanings than that which is used here.
## Authors

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob Praus</td>
<td><a href="mailto:praus@mza.com">praus@mza.com</a></td>
</tr>
<tr>
<td>Steve Coy</td>
<td><a href="mailto:coy@mza.com">coy@mza.com</a></td>
</tr>
<tr>
<td>MZA Associates Corporation</td>
<td><a href="http://www.mza.com">www.mza.com</a></td>
</tr>
</tbody>
</table>
Acknowledgments

Building on broader concepts in the technical communities, the fundamental ideas in tempus have been in development for more than two decades. Steve Coy is the primary designer and authored the current distribution version. Bob Praus helped write tempus and has applied it more than anyone. A lot of people have helped along the way.

Don Washburn, Russ Butts & Roy Hamil
Gregory Gershanok
Ali Boroujerdi & Steve Verzi
Zane Dodson
Bill Klein
Alex Zokolov
Liyang Xu, Tim Berkopec, Boris Venet
Robert Suizu, Brent Strickler, Bill Gruner, Keith Beardmore, Justin Mansell, Morris Maynard, & Tony Seward

AFRL funding & encouragement
GUI developer and code integrator
Authors of newer kernel prototypes
Design and authoring of advanced features
Design assessment & code integration
Advanced GUI and visualization development
Developers and users of tempus systems

We would also like to thank the DEPS for providing this forum.
References

• Modeling and Simulation
  – http://www.dmem.strath.ac.uk/~pball/simulation/simulate.html
  – http://www.ecs.umass.edu/ece/labs/codes/bktoc.html
  – http://www.ici.ro/ici/revista/sic2002_1/art05.htm

• Object-oriented Programming

• The C++ Programming Language
  – http://www.cppforlife.tk/
<table>
<thead>
<tr>
<th>Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modeling and simulation concepts</strong></td>
</tr>
<tr>
<td>Time-domain and discrete event modeling</td>
</tr>
<tr>
<td>Composition-based modeling</td>
</tr>
<tr>
<td>Isomorphic modeling</td>
</tr>
<tr>
<td>Multi-modeling</td>
</tr>
<tr>
<td>Simulation executives</td>
</tr>
<tr>
<td>Object-oriented modeling in C++</td>
</tr>
<tr>
<td><strong>The tempus paradigm</strong></td>
</tr>
<tr>
<td>Overview</td>
</tr>
<tr>
<td>tempus visual editor</td>
</tr>
<tr>
<td>tempus concepts</td>
</tr>
<tr>
<td>Connection-driven execution</td>
</tr>
<tr>
<td>tempus source form</td>
</tr>
<tr>
<td><strong>The future of tempus</strong></td>
</tr>
</tbody>
</table>
Modeling and Simulation Concepts

- Time-domain and discrete event modeling
- Composition-based modeling
- Isomorphic modeling
- Multi-modeling
- Simulation executives
- Object-oriented modeling in C++
**Simulation**: the technique of imitating the behavior of some situation or system (economic, mechanical, etc.) by means of an analogous model, situation, or apparatus, either to gain information more conveniently or to train personnel. (Oxford Eng. Dictionary)

**Time-domain modeling**: a technique in which the performance of a system is simulated by predicting the state of the system as a function of time.

**Discrete event-driven modeling**: a time-domain simulation technique in which the logic of the simulation is primarily governed by specific events which occur within the modeled system.
• **Composition-based modeling**: the process of building software models by combining smaller, more fundamental, software components.

• **Multi-modeling**: the use of composition-based modeling in interdisciplinary physical modeling problems.

• **Variable fidelity modeling**: the process of building and employing a model which has multiple levels of fidelity.

• **Isomorphic**: exactly corresponding in form and relations. (Oxford Eng. Dictionary)

• **Isomorphic modeling**: the design and implementation of a model using isomorphism as a prevailing guiding principle.
## Simulation Executives

- Software tools meant to assist in the development and use of simulations.
- Usually specific to a particular domain.
- One would rarely use the simulation executive if one were not interested in the particular domain to which the simulation executive applies.
- Generally not appropriate for large simulations.
- Usually composition-based.
- Methods to expand the library of models is limited.
- Component behavior is usually limited to a particular fundamental modeling approach.
- Examples: Simulink, Easy5, acslXtreme, Systembuild, SPICE
Modeling and Simulation Concepts

<table>
<thead>
<tr>
<th>Time-domain and discrete event modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition-based modeling</td>
</tr>
<tr>
<td>Isomorphic modeling</td>
</tr>
<tr>
<td>Multi-modeling</td>
</tr>
<tr>
<td>Simulation executives</td>
</tr>
<tr>
<td>Object-oriented modeling in C++</td>
</tr>
</tbody>
</table>
Object-Oriented Programming

- A computer programming paradigm in which a program is based on a collection of individual units, or objects, that act on each other, as opposed to a traditional (procedural) paradigm in which a program may be seen as a collection of functions or procedures, or simply as a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects. (Wikipedia)
Object Oriented Programming
The Benefits

• Benefits of OOP
  – Facilitates the application of isomorphism - the programming practice of implementing a one-to-one correspondence between segments of code and modeled entities.
  – Facilitates modularity of both code and data.
  – Facilitates the application of polymorphism - the programming practice of using the same code for different objects which have common characteristics.
  – High-level (executive) code is highly readable.

• Benefits of C++
  – Both widespread and highly supported.
  – Very efficient (largely because it is based on C).
  – Supports the implementation of both high-level (executive) and low-level (math and bit-twiddling) code.

There are a lot of advantages to OOP. See Object-Oriented Analysis and Design by Grady Booch for more complete information.
Object Oriented Programming
The Perils (because you can)

• Perils of OOP
  – OOP codes are susceptible to over-design -- churning over the design of a particular feature without any real benefit (because you can).
  – OOP codes are susceptible to over-implementation -- coding an object such that it can do any conceivable operation (because you can) when all that is really necessary is meeting current requirements. This results in wasted effort and a legacy of untested code because many routines are never used.
  – As a result of the two previous susceptablities, OOP codes can become spaghetti codes of a new sort. This particular form of tangularity results in practically every line of code being a reference to code in some other compilation unit. Finding bugs then involves a lot of unnecessary hopping around between source files.
  – Programmers can mistakenly rely on the OOP model as a substitute for true innovation (because you can).

• Perils of C++
  – C++ arrays are inflexible (especially multi-dimensional arrays). For mathematical codes, this results in having to implement a substitute.
  – C++ pointers are dangerous. Memory leaks and dangling pointers are common.
  – C++ templates can be bad. Don’t use them unless you know what you are doing.
  – C++ has obtuse syntax. Low-level code can be difficult to read.

Despite these dangers, using OOP within C++ is probably the most flexible and powerful contemporary approach to developing a complex application which is both portable and efficient.
Classes, base classes and virtual methods are all standard terms used in object-oriented programming.

A class is language-level construct which can be used to encapsulate a well-defined software representation of a specific category of objects, including both its data members and its behavior.

A class can inherit attributes (data and/or behavior) from one or more other classes, called its base classes. Some classes, like System in tempus, are specifically designed to be used as base classes.

Virtual methods are “stub” functions defined in a base class which can be re-defined by derived classes. Virtual methods are used to define standardized interfaces for customizable behaviors.
C++ Templates

• Templates are a way of implementing C++ functions and classes in a type-neutral kind of way.

• The Type of interest is specified to the Template code at compile time and the appropriate code is generated taking into account fairly generic aspects of the underlying type.

• This is how one might implement a vector of integers with essentially the same code as they might implement a vector of floats.

• Templates can also be used to specify other compile-time attributes.
Base class:

class TRect {
public:
    // data members
    short fTop;
    short fLeft;
    short fBottom;
    short fRight;
    // member functions
    virtual short Area(void);
    Boolean PointInRect(Point thePt);
};

Class which uses inheritance:

class TRoundRect : public TRect {
protected:
    // added data members
    short fHOval;
    short fVOval;
    // override the area member function
    virtual short Area(void);
};

Template class:

template<class T> class vector {
    T* v;
    int sz;
public:
    vector (int);
    T& operator[] (int);
    T& elem(int i) { return v[i]; } 
    // ...
};

Using Classes:

int top, left, bottom, right;
...
TRect r(top, left, bottom, right);
TRoundRect rr(top, left, bottom, right);

vector<float> vf(5);
vf[0] = (r.area() + rr.area())/2.0;

vector<int> vi(4);
vi[0] = top;
vi[1] = left;
vi[2] = bottom;
vi[3] = right;
The **tempus** Paradigm

Overview
- tempus visual editor
- tempus concepts
- Connection-driven execution
- tempus source form
tdemo1 Example

- To provide context, we'll go to a short demonstration the creation, execution, examination, and manipulation of a simple tempus user application.
The process supports tempus System development, debugging, and analysis.
```c
#include "tempus.h"
#include "TopLevel.h"

Notional Generated Code

int main(int argc, char* argv[]) {
    Universe u(NULL, "u");
    int p1 = 2;
    double p2 = 3.1415;
    for (iloop=0; iloop<nloop; iloop++) {
        double p3 = iloop * p1 * p2;
        TopLevel t(u, "t", p1, p2, p3);
        Recorder r(u, "r");
        r.i <= t.ss.o;
        u.advanceTime(stopTime);
    }
}

class TopLevel : public System {
public:
    int p1;
    double p2;
    double p3;
    Subsystem ss;
    TopLevel(System* p, char* n, int _p1,...) :
        System(p, n),
        p1(_p1), p2(_p2), p3(_p3), ss(p1, p2, p3) {
    ...
    }
};
```
The **tempus** Paradigm

Overview

- tempus visual editor
- tempus concepts
- Connection-driven execution
- tempus source form
## tempus Visual Editor

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tve</strong> – tempus visual editor</td>
<td>The graphical user interface (GUI) through which the user constructs tempus models and sets up and executes tempus simulations. Given user inputs, the tve generates code which is compiled and linked with user-written code to create and execute user applications.</td>
</tr>
<tr>
<td><strong>tse</strong> – tempus system editor</td>
<td>The tve window used to create, configure, and edit tempus Systems.</td>
</tr>
<tr>
<td><strong>tre</strong> – tempus runset editor</td>
<td>The tve window used to set up and execute tempus simulations.</td>
</tr>
</tbody>
</table>
tve & tse
### Run Variables

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double</td>
<td>stopTime</td>
<td>0.025 Simulate stop time</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>case</td>
<td>3 Stop(3) General loop variable</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>rand</td>
<td>0 Atmospheric randomization loop variable</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>control</td>
<td>0 Controls parameterization loop variable</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>range0</td>
<td>0 Start range to target (m)</td>
</tr>
<tr>
<td>6</td>
<td>Vector</td>
<td>vplane</td>
<td>TwoVec(0,0,0) Aircraft velocity (m/s)</td>
</tr>
<tr>
<td>7</td>
<td>Vector</td>
<td>vtarget</td>
<td>TwoVec(0,0,175) Target velocity (m/s)</td>
</tr>
<tr>
<td>8</td>
<td>float</td>
<td>Dap0</td>
<td>1.5 Telescope aperture diameter (m)</td>
</tr>
</tbody>
</table>

### System Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>float</td>
<td>range0</td>
<td>Range to beacon/target (m)</td>
</tr>
<tr>
<td>2</td>
<td>float</td>
<td>tel_focus_range</td>
<td>Focal distance of telescope (m)</td>
</tr>
<tr>
<td>3</td>
<td>AcScAlMSpec</td>
<td>AlmSpec</td>
<td>TwoVec(0,0,0),distances,strengths,specification of atmosphere. AcScAlMSpec(wavelength)</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>AlmSeed</td>
<td>Random seed for phase screens</td>
</tr>
<tr>
<td>5</td>
<td>float</td>
<td>tmax</td>
<td>Maximum length of time used to size phase screens</td>
</tr>
<tr>
<td>6</td>
<td>Vector</td>
<td>vplane</td>
<td>Platform velocity (x,y in m/s)</td>
</tr>
<tr>
<td>7</td>
<td>Vector</td>
<td>vtarget</td>
<td>Target velocity (x,y in m/s)</td>
</tr>
<tr>
<td>8</td>
<td>Vector</td>
<td>vwind</td>
<td>Wind velocity assumed uniform throughout (x,y, m/s)</td>
</tr>
<tr>
<td>9</td>
<td>DMModel</td>
<td>dmModel</td>
<td>Specification of DM geometry</td>
</tr>
<tr>
<td>10</td>
<td>float</td>
<td>Dap0</td>
<td>Diameter of telescope aperture (m)</td>
</tr>
<tr>
<td>11</td>
<td>float</td>
<td>tel_wavelength</td>
<td>Wavelength of outgoing laser (m)</td>
</tr>
<tr>
<td>12</td>
<td>float</td>
<td>beacon_wavelength</td>
<td>Wavelength of incoming point source (m)</td>
</tr>
<tr>
<td>13</td>
<td>float</td>
<td>link_wavelength</td>
<td>Wavelength of auxiliary point source (m)</td>
</tr>
<tr>
<td>14</td>
<td>float</td>
<td>beacon_x</td>
<td>X location of the beacon (m)</td>
</tr>
<tr>
<td>15</td>
<td>float</td>
<td>beacon_y</td>
<td>Y location of the beacon (m)</td>
</tr>
<tr>
<td>16</td>
<td>int</td>
<td>npropy</td>
<td>Number of grid points on the propagation grid</td>
</tr>
<tr>
<td>17</td>
<td>float</td>
<td>npropoy</td>
<td>Propagation grid spacing (m)</td>
</tr>
<tr>
<td>18</td>
<td>int</td>
<td>tbd_ray</td>
<td>Number of pixels on the targetboard</td>
</tr>
</tbody>
</table>
The **tempus** Paradigm

- Overview
- tempus visual editor
- tempus concepts
- Connection-driven execution
- tempus source form
Variables, Types, and Names

• Systems, Inputs, Outputs, and Parameters are implemented as programming variables and all have types and names.

• Type refers to the particular data type of the entity. In this usage, type and class are nearly synonymous.
  – All Systems are of some Type which must be derived from class System. So all Systems are classes.
  – All Inputs and Outputs have a type, but not through inheritance. Rather, Inputs and Outputs get their type through a template argument. The type can be just about any valid C++ type, but it must support a few standard operations. The GUI hides the details concerning the use of templates.
  – Parameters are simple variables of a user-specified type. The type which can be simple, such as float or int or more complex, such as an arbitrary class. The type can be just about any valid C++ type, but it must support a few standard operations.

• Name refers to the name of the particular variable within the context that it resides.
Types and names in the tve
### tempus Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System</strong></td>
<td>The base class for the fundamental building block of tempus applications. Specific Systems can be automatically generated or user-written. Systems are configured by their Parameters and contain Inputs and Outputs in facilitate time-domain interfaces with other Systems.</td>
</tr>
<tr>
<td><strong>Input&lt;T&gt;</strong></td>
<td>The primary mechanism through which a System is affected by other Systems.</td>
</tr>
<tr>
<td><strong>Output&lt;T&gt;</strong></td>
<td>The primary mechanism through which a System can effect other Systems.</td>
</tr>
<tr>
<td><strong>Universe</strong></td>
<td>A top-level executive object which controls the order of System execution and the passage of time.</td>
</tr>
</tbody>
</table>
## Categories of Systems

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem</td>
<td>A <strong>System</strong> contained in another <strong>System</strong>. Almost all <strong>Systems</strong> are <strong>Subsystems</strong> because all <strong>Systems</strong>, except the very top-level <strong>System</strong>, is contained by another.</td>
</tr>
<tr>
<td>Composite System</td>
<td>A <strong>System</strong> composed of one or more <strong>Systems</strong>. <strong>Composite Systems</strong> are typically (but don't have to be) generated by the <strong>tempus system system editor</strong>.</td>
</tr>
<tr>
<td>Atomic System</td>
<td>A <strong>System</strong> written to carry-out computations of specific interest. Ultimately, all of the meaningful computation of a tempus user application is done by an <strong>Atomic System</strong>.</td>
</tr>
<tr>
<td>Top-level System</td>
<td>A <strong>System</strong> which contains all other <strong>Systems</strong> in a particular tempus user application.</td>
</tr>
</tbody>
</table>
## tempus Parametric Concepts

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter</strong></td>
<td>The mechanism through which Systems are configured. The values of Parameters are provided to Systems through constructor arguments, so they are only effective in specifying static initialization inputs. Parameters of Subsystems are often specified by expressions involving Parameters of the System that contains them.</td>
</tr>
<tr>
<td><strong>Runset</strong></td>
<td>The collection of information which specifies the Parameter values for a set of user application executions. The Runset specifies the Parameter values for the Top-Level System which Outputs are to be recorded.</td>
</tr>
</tbody>
</table>
tempus Concepts
System Parameters
tempus Concepts
System Parameters Flow Down from Containing Systems
Runsets...

- define the values of all parameters which the model-builder has "flowed-up" to the user.

- provide a configuration management tool for defining the inputs of a run.

- are used to set up parametric studies, allowing parameters to be changed systematically.

- definitions help to define how work is distributed across multiple processors.
tempus concepts

Block Parameters Flow Down from Runset
# Concepts Not Detailed in This Course

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recallability</td>
<td>The mechanism through which Systems can request the values of their input for some time in the past.</td>
</tr>
<tr>
<td>Recalling&lt;T&gt;</td>
<td>The class through which Recallability is implemented.</td>
</tr>
<tr>
<td>SaveVariable&lt;T&gt;</td>
<td>Another class which helps implement Recallability.</td>
</tr>
</tbody>
</table>
The **tempus** Paradigm

Overview
- tempus visual editor
- tempus concepts
- Connection-driven execution
- tempus source form
tempus Concepts
Inputs, Outputs, and Connectivity
**Input and Output Types**

- Inputs and Outputs are template-typed classes.
- Inputs and Outputs can be of nearly any valid C++ type.
- Connections are only made between two entities of the same type.
- Provisions have been made to provide automatic conversions between types which are nearly compatible.
### Three Types of Connections

<table>
<thead>
<tr>
<th>Connection</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Subsystem Output to Subsystem Input**  
\[ \text{ss1.i} \ll= \text{ss2.o} \] | The most intuitive type of connection feeds a *Subsystem's Output* to a *Subsystem's Input*. |
| **Composite System Input to Subsystem Input**  
\[ \text{ss.i} \ll= \text{i} \] | *Composite System Inputs* are routed to its *Subsystem Inputs*. |
| **Subsystem Output to Composite System Output**  
\[ \text{o} \ll= \text{ss.o} \] | *Subsystem Outputs* can become *Outputs* of the containing Composite System. |
# Default Behaviors

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value assigned to Input</td>
<td>Default values for an Input can be specified so that the Input does not have to be externally connected.</td>
</tr>
<tr>
<td>Default Value assigned to System Output</td>
<td>Default values for System Outputs can be specified to provide an output value in the situation that a Subsystem Output is not eventually connected to it.</td>
</tr>
</tbody>
</table>
Connections in the tve

Note: A Subsystem Input cannot have two connections
Connection-driven Execution

main

Universe

advanceTime

scheduleNotice
scheduleEvent
cancelEvent

respondToWarning
respondToChangedInputs
respondToScheduledEvent

System

respondToWarning
respondToInputWarning
respondToOutputRequest

Output

Input

Output

Input

value

attempt to access

attempt to modify

value
class System : public SystemNode
{
    protected:
        virtual void respondToInputWarning(InputBase* input);
        virtual void respondToChangedInputs();
        virtual void respondToOutputRequest(const OutputBase* /*output*/);
        virtual void respondToScheduledEvent(const Event& /*event*/);
    ...
};

• Depending on the desired system behavior, the Atomic System coder writes a System-specific implementation of one or more virtual methods.

• Composite Systems do not implement the virtual methods because the behavior of Composite Systems is governed by the behavior its Subsystems.

• Each of the virtual methods have default logic so that Atomic Systems do not have to overload methods unrelated to its desired execution behavior.
Input-Driven Logic

class System : public SystemNode
{
    protected:
        virtual void respondToInputWarning(InputBase* input);
        virtual void respondToChangedInputs();
        virtual void respondToOutputRequest(const OutputBase* /*output*/);
        virtual void respondToScheduledEvent(const Event& /*event*/);

    ...
}

• `respondToInputWarning (InputBase*)` warns a System that one of its Inputs is about to be changed.

• Before any System changes an Output which is connected to another System's Input, the Input's System's `respondToInputWarning (InputBase*)` is called.

• `respondToChangedInputs ()` notifies a System that one or more of it's inputs has been changed.
Output-Driven Logic
(Lazy Evaluation)

class System : public SystemNode
{
    protected:
        virtual void respondToInputWarning(InputBase* input);
        virtual void respondToChangedInputs();
        virtual void respondToOutputRequest(const OutputBase* /*output*/);
        virtual void respondToScheduledEvent(const Event& /*event*/);
        ...
}

• When a System accesses the value of an Input which is connected to another System's Output, that Output's System's respondToOutputRequest (OutputBase) is called.
Event Driven Logic

class System : public SystemNode
{
    protected:
        virtual void respondToInputWarning(InputBase* input);
        virtual void respondToChangedInputs();
        virtual void respondToOutputRequest(const OutputBase* /*output*/);
        virtual void respondToScheduledEvent(const Event& /*event*/);
    private:
        EventId scheduleEvent(double delay, char* descriptor="", void* info=NULL)
        ...
}

• A **System** can exercise strong control over its execution by scheduling **Events** for itself by invoking the `scheduleEvent(...)` method.

• After the specified amount of time has passed, the scheduler called the System's `respondToScheduledEvent(const Event&)` method.
Complex Producer-Consumer Models

At time $t$, the receiver asks the next component upstream to tell it about the light incident upon it. Each intervening component asks the next component upstream to tell it about what light is incident upon it. Each light source must be prepared to describe the light transmitted from it using one or more “waves.” It must provide certain info about itself: aperture size and location, field of view, wavelengths sensed, etc. It must provide information about receiver and the optical path between it and the receiver. It must take into account the information provided about receiver and the optical path between it and the receiver. The source constructs the first wave, then returns.

Each intervening component operates on the wave, then returns. The source maps the wave to its detector plane. Each intervening component asks the next component upstream for the next wave incident upon it. When the source has no more waves to send, it returns a NULL. Each intervening component then returns a NULL. When the receiver receives a NULL, it knows it has received all the waves incident upon it at time $t$. The source then checks whether it needs to send any more waves.

Inputs and Outputs can be of nearly any valid C++ type. The extreme flexibility of connection-driven execution combined with sophisticated Input-Output types, can provide extremely complex System interactions. MZA's wave-optics code is named after its fundamental interface type, WaveTrain, which provides a two-way dialog between optical components.
The tempus Paradigm

Overview
- tempus visual editor
- tempus concepts
- Connection-driven execution
- tempus source form
Code Generation Strategy

- Atomic Systems are built by the tve as System class stubs.
  - The programmer is expected to implement virtual methods which define the System's behavior.
  - Because many systems have common features, inheritance and polymorphism is used a lot.
- Composite Systems are coded as complete Systems
  - Parameters are constructor arguments.
  - Inputs and Outputs are member objects.
  - Subsystems are declared and initialized using expressions involving the parameters of the system.
  - Subsystems are connected using the simple overloaded operator $\ll=$.
  - Miscellaneous code handles default unconnected inputs.
- Runsets are coded as the main program.
  - The code contains explicit loops for loop variable.
  - The run variables and top-level system parameters are declared and set. Run variables and system parameters which are dependent on loop variables inside the appropriate loops.
  - The top-level system is constructed using the system parameters.
  - Recording systems are constructed and connected.
  - Each run is executed with a call to advanceTime(…).
  - There is miscellaneous code which takes care of runset monitoring and setting up the output trf file.
Gain : An Atomic System

```cpp
class Gain : public System {
    private:
        float k;
    public:
        Input<float> u;
        Output<float> y;
        Gain(SystemNode* parent, char* name, float _k) :
            System(this, name),
            k(_k), u(this, "u"), y(this, "y") {}
    private:
        void respondToInputWarning(InputBase* input) {
            y.warnReferencers();
        }
        void respondToOutputRequest() {
            y=k*u;
        }
};
```

Atomic Systems' code is written by hand. In this case, the code in blue is all the logic that was added. The GUI provided the rest in the form of a template.

DoubleGain : A Composite System

```cpp
class DoubleGain : public System {
    private:
        Gain gain1;
        Gain gain2;
    public:
        Input<float> u;
        Output<float> y;
        DoubleGain(SystemNode* parent, char* name, float _k1, float _k2) :
            System(this, name),
            gain1(this, "gain1", _k1),
            gain2(this, "gain2", _k2),
            u(this, "u"), y(this, "y") {
                gain1.u <<= u;
                gain2.u <<= gain1.y;
                y <<= gain2.y;
            }
};
```

Composite Systems' code is usually generated by the GUI.
tempus SquareWave Example

Atomic system SquareWave uses event-driven logic.

class SquareWave : public System {
private:
    float pulseLength;
    float pulseInterval;
public:
    Output<float> y;
    SquareWave(SystemNode parent, char* name,
                float _pulseLength,
                float _pulseInterval,
                float _delay) :
        SystemNode(parent, name),
        pulseLength(_pulseLength),
        pulseInterval(_pulseInterval),
        y(this, "y")
        {
            scheduleEvent(_delay,"begin pulse");
        }
    
private:
    void respondToScheduledEvent(const Event& event)
    {
        if (event == "beginPulse")
        {
            y = 1.0;
            scheduleEvent(pulseLength, "end pulse");
            scheduleEvent(pulseInterval, "begin pulse");
        }
        else if (event == "end pulse")
        {
            y = 0.0;
        }
    }
};

The code in blue was written by the System implementer. The rest of the code was provided by the GUI as a template.
tempus Main Program Example

```c
main()
{
    SquareWave sw(0.1, 0.5, 0); // construct a Square Wave.
    Gain g(2.0); // construct a Gain
    Sampler<float> s(); // construct a Sampler.
    g.u <<= sw.y; // connect Gain’s input to the SquareWave’s output
    s.u <<= g.y; // connect the Sampler’s input to the Gain’s output
    advanceTime(100.0); // advance virtual time 100 seconds
}
```

- The main program is usually generated by the GUI, but it can be written by hand just as well.
tempus Code is Readable

● Each composite system declares and initializes its subsystems:

```plaintext
pointsorce(this, "pointsorce", wavelength, 1.0e6, 0.0, 0.0),
transversevelocity1(this, "transversevelocity1", -wind, 0.0, 0.0, 0.0),
transversevelocity3(this, "transversevelocity3", wind, 0.0, 0.0, 0.0),
atmosphericpath1(this, "atmosphericpath1",
AcsAtmSpec(wavelength, nscreen, clear1Factor, hPlatform, hTarget, range),
atmoSeed, propnxy, propdxy, 1.8, 0.05,
-propnxy*propdxy/2.0, propnxy*propdxy/2.0, -propnxy*propdxy/2.0, propnxy*propdxy/2.0,
-propnxy*propdxy/2.0, propnxy*propdxy/2.0, -propnxy*propdxy/2.0, propnxy*propdxy/2.0,
propdxy, 0.0, 0.0, 0),
camera1(this, "camera1", 1.0, wavelength, wavelength, apdiam/propdxy,
propdxy, 64, wavelength/apdiam, 0.0),
simplefieldsensor1(this, "simplefieldsensor1", wavelength, apdiam/propdxy, propdxy),
telescope1(this, "telescope1", range, apdiam/2.0, 0.0),
incomingsplitter1(this, "incomingsplitter1")
```

● Then the subsystems inputs and outputs are connected:

```plaintext
simplefieldsensor1.incident <<= incomingsplitter1.incomingTransmitted2;
camera1.incident <<= incomingsplitter1.incomingTransmitted;
incomingsplitter1.incomingIncident <<= telescope1.incomingTransmitted;
telescope1.incomingIncident <<= transversevelocity3.incomingTransmitted;
atmosphericpath1.incomingIncident <<= transversevelocity1.incomingTransmitted;
transversevelocity1.incomingIncident <<= pointsource.transmitted;
```

● Then the simulation is run:

```plaintext
advanceTime(stopTime);
```

blue names are systems
green names are inputs
red names are outputs
cyan names are regular variables
A Complete tempus Run

```c
#include "tempus.h"
#include "Recorders.h"
#include "FileSys.h"
#include "PointSource.h"
#include "AtmoPath.h"
#include "Telescope.h"
#include "Camera.h"

#ifndef NO_TEMPUS_SMF_MONITOR
#include "TempusStatusSMF.h"
#endif

main(int argc, char* argv[])
{
    // Decoration related to monitoring the system during the run.
    //
    // ifndef NO_TEMPUS_SMF_MONITOR
    double stopTime = 0.0050;
    char ***outfile = "WtDemoRunHand.trf";
    char ***trfname;
    char ***smfname;
    parseName(argc, argv, ***outfile, ***smfname, ***trfname, stopTime);
    TempusStatusSMFWriter ***smfWriter(***smfname, ***trfname, "", 1);
    setCurrentSMF(***smfWriter);
    #endif

    Universe ut1("Hand");
    // Construction of all the systems. Variables could be used in the parameters
    // below rather than the constants.
    //
    PointSource pointsource(NULL, "ps", 1.0e-06, 1.0e+06, 0.0, 0.0);
    AtmoPath atmosphericpath(NULL, "ap", 
        AcsAtmSpec(1.0e-06,10.2,0.2413,0.2728,0.52600.0), 
        -765432189, 256, 0.0, 1.8, 0.05, 
        -256*0.02/2.0, 256*0.02/2.0, -256*0.02/2.0, 256*0.02/2.0, 
        -256*0.02/2.0, 256*0.02/2.0, 256*0.02/2.0, 256*0.02/2.0, 
        0.02, 0.0, 0.0, 0);
    Telescope telescope(NULL, "tel", 52600.0, 1.5/2.0, 0.0);
    Camera camera(NULL, "cam", 1.0, 1.0e-06, 1.0e-06, 1.5/0.02, 0.02, 64, 
        1.0e-06/1.5, 0.0);

    // Connection of the systems.
    //
    atmosphericpath.incomingIncident <<= pointsource.transmitted;
    telescope.incomingIncident <<= atmosphericpath.incomingTransmitted;
    camera.incident <<= telescope.incomingTransmitted;
    //
    // Construction and connection of non-connected inputs.
    //
    Output<bool> camera_on(&camera, "cam_on", true);
    Output<double> camera_ei(&camera, "cam_el", 1.0e-3);
    Output<double> camera_el(&camera, "cam_el", 1.0e-6);
    Output<double> camera_si(&camera, "cam_el", -1.0);
    camera.on <<= camera_on;
    camera.exposureInterval <<= camera_ei;
    camera.exposureLength <<= camera_el;
    camera.sampleInterval <<= camera_si;
    //
    // Decoration related to recording the outputs.
    //
    ParamSet pst1;
    RecorderFile rft1(NULL, "rft1", ***trfname, ParamSet_stringify(pst1), 
        pst1);
    GridRecorder<float> rft11(NULL, "rft11", "camera.fpalimage", 
        "Grid<float>", "image", true, (float)0.0, 0.0);
    rft11.dr <<= rft1.dr;
    rft11.i <<= camera.fpalimage;
    //
    // Run the simulation.
    //
    advanceTime(stopTime);
}
```

// To run:
//    setupwt
//    mktr WtDemoRunHand
//    WtDemoRunHand
The Future of tempus

Continuous Time Dynamics Solver
Dynamic System Composition
Multi-Inputs and Multi-Outputs
Heavy use of stl
Runtime inspection & modification
New GUI
Continuous Time Dynamics Solver

• tempus 2006 has been upgraded to include a powerful DAE solver to provide for the solution of continuous time dynamics.

• The following pages show a planar seven body problem called "The Pleiades" as implemented and tested in tempus 2006.


• Zane Dodson, a consultant to MZA, implemented the tempus continuous time solver and The Pleiades solution which follows.
class GravitationalForce : public tSystem
{
    public:
        GravitationalForce(const string& name = "", double G = 0.0)
            : tSystem(name), G(G), body1("body1"), body2("body2"),
                force_on_1_by_2("force_on_1_by_2"), force_on_2_by_1("force_on_2_by_1")
            {
                add(&body1);
                add(&body2);
                add(&force_on_1_by_2);
                add(&force_on_2_by_1);
            }
            virtual void respondToOutputRequest(const tOutput*)
            {
                tV2 displacement = body2.get().position - body1.get().position;
                const double distance = norm(displacement);
                const tV2 f = (G * body1.get().mass * body2.get().mass * displacement
                                / (distance * distance * distance));
                force_on_1_by_2.set(f);
                force_on_2_by_1.set(-1.0 * f);
            }
    tInputT<BodyDynamics> body1;
    tInputT<BodyDynamics> body2;
    tOutputT<tV2> force_on_1_by_2;
    tOutputT<tV2> force_on_2_by_1;
    private:
        double G;
};
class Body : public tSystem
{
public:
    Body(const string& name = "", double mass = 0.0, const tV2& r0 = tV2(),
         const tV2& rdot0 = tV2())
        :
            tSystem(name), force("force", true), dynamics("dynamics"), mass(mass),
            r0(r0), rdot0(rdot0)
    {
        add(&force);
        add(&dynamics);
        r.setContainer(this); // FIXME
        rdot.setContainer(this); // FIXME

        const double nan = numeric_limits<double>::quiet_NaN();
        const tV2 rddot0 = tV2(nan, nan);
        r.set(r0, rdot0);
        rdot.set(rdot0, rddot0);

        tVariable::addDependency(&force, &rdot.residual());
        tVariable::addDependency(&r, &dynamics);
        tVariable::addDependency(&rdot, &dynamics);
        tVariable::addDependency(&rdot.derivative(), &dynamics);
    }
};
void init() // FIXME
{
    tV2 cummulative_force(0.0, 0.0);
    for (tInputT<tV2>::iterator i = force.begin(); i != force.end(); ++i)
        cummulative_force += *i;
    rdot.set(rdot0, cummulative_force / mass);
}

virtual void respondToComputeOde(const tContinuousState* state)
{
    if (state == &r)
        r.residual().set(rdot.get() - r.derivative().get());
    else
    {
        tV2 cummulative_force(0.0, 0.0);
        for (tInputT<tV2>::iterator i = force.begin(); i != force.end(); ++i)
            cummulative_force += *i;
        rdot.residual().set(cummulative_force - mass * rdot.derivative().get());
    }
}

virtual void respondToOutputRequest(const tOutput*)
{
    dynamics.set(BodyDynamics(r.get(), rdot.get(), rdot.derivative().get(), mass));
}

tInputT<tV2> force;
tOutputT<BodyDynamics> dynamics;

private:
    double mass;
    tV2 r0, rdot0;
    tContinuousStateT<tV2> r, rdot;
int main()
{
    const double G = 1.0;
tUniverse U("U");

    std::vector<Body*> bodies;
bodies.push_back(new Body("body1", 1.0, TV2(3.0, 3.0), TV2(0.0, 0.0)));
bodies.push_back(new Body("body2", 2.0, TV2(3.0, -3.0), TV2(0.0, 0.0)));
bodies.push_back(new Body("body3", 3.0, TV2(-1.0, 2.0), TV2(0.0, 0.0)));
bodies.push_back(new Body("body4", 4.0, TV2(-3.0, 0.0), TV2(0.0, -1.25)));
bodies.push_back(new Body("body5", 5.0, TV2(2.0, 0.0), TV2(0.0, 1.0)));
bodies.push_back(new Body("body6", 6.0, TV2(-2.0, -4.0), TV2(1.75, 0.0)));
bodies.push_back(new Body("body7", 7.0, TV2(-2.0, 4.0), TV2(-1.50, 0.0)));

    for (int i = 0; i < bodies.size(); ++i)
        U.add(bodies[i]);

    std::vector< std::vector<GravitationalForce*> > gf(bodies.size(),
        std::vector<GravitationalForce*>(bodies.size()));

    for (int i = 0; i < bodies.size(); ++i)
        for (int j = i+1; j < bodies.size(); ++j)
        {
            gf[i][j] = new GravitationalForce("", G);
            U.add(gf[i][j]);
            gf[i][j]->body1.connect(&bodies[i]->dynamics);
            gf[i][j]->body2.connect(&bodies[j]->dynamics);
            bodies[i]->force.connect(&gf[i][j]->force_on_1_by_2);
            bodies[j]->force.connect(&gf[i][j]->force_on_2_by_1);
        }

    ...
for (int i = 0; i < bodies.size(); ++i)
    bodies[i]->init();

for (int k = 0; k <= 300; ++k)
{
    cout << U.now();
    for (int i = 0; i < bodies.size(); ++i)
        cout << "\t" << bodies[i]->dynamics.get().position;
    for (int i = 0; i < bodies.size(); ++i)
        cout << "\t" << bodies[i]->dynamics.get().velocity;
    for (int i = 0; i < bodies.size(); ++i)
        cout << "\t" << bodies[i]->dynamics.get().acceleration;
    cout << endl;
    U.tick(0.01);
}

...
Pleiades Solution
The positions of 7 stars traced in a plane


Z. Dodson, tempus 2006 – Continous Time DAE Solver
The New tempus GUI