A C++ Instance of TeD

Kalyan S. Perumalla and Richard M. Fujimoto
(kalyan@cc.gatech.edu and fujimoto@cc.gatech.edu)
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

GIT-CC-96-33
December 31, 1996
(last updated January 28, 1997)

ABSTRACT

TeD is a language designed mainly for modeling telecommunication networks. The TeD language specification is separated into two parts — (1) a meta language (2) an external language. The meta language specification is concerned with the high-level description of the structural and behavioral interfaces of various network elements. The external language specification is concerned with the detailed low-level description of the implementation of the structure and behavior of the network elements. The meta language, called MetaTeD, is described in a separate related document. An external language specification, with C++ as the external language, is described in this document. A software support system for the development and parallel simulation of TeD models is also described.
Contents

1 Introduction 2

2 Language Restrictions 3

3 Macro Interface 4
   3.1 Macro Summary ................................................. 4
   3.2 Macro Descriptions ............................................. 6

4 Configuration Language 17
   4.1 Variable Definition ............................................ 17
   4.2 For Statement .................................................. 18
   4.3 Require Statement .............................................. 20
   4.4 Preprocessing .................................................. 20

5 Compiler 21

6 Support Libraries 21

7 Support Tools 22
   7.1 Model Dependency Generator .................................. 23
   7.2 Resource Database Generator .................................. 23
   7.3 Model Information Collector ................................... 23
   7.4 Platform-independent System Builder ......................... 23

8 Deviations from MetaTeD 24

9 Model Development Guidelines 25

A Example 1: Pingpong 27

B Example 2: ATM Multiplexer 32
1 Introduction

TeD is a language for modeling telecommunication networks. This document describes a C++ “instance” of the TeD language specification (see [1]), as well as a Time Warp-based simulation support system for analyzing TeD models that are developed using the C++ interface.

In the rest of the document, all references to “TeD models” are to those TeD models that are developed using the C++ language interface to TeD.

The software system for TeD model development and simulation can be obtained by sending email to the authors. The software version number as of this writing is TED1.5. The system consists mainly of the following:

- **Language Restrictions**: A set of C++ language elements and operations that are disallowed to be used in the external code of TeD models. These together, in effect, define a strict subset of the C++ language that can be used in the TeD models. Section 2 lists these restrictions.

- **Macro Interface**: A set of macros that can be used in the external code blocks of TeD model descriptions. Section 3 describes the various macros, their semantics and usage.

- **Configuration Language**: A language using which any configuration of models can be assembled and customized for instantiation and simulation. The configuration language specification is presented in Section 4.

- **Compiler**: A compiler for translating TeD models to target simulation code. Section 5 describes the implementation and usage of the compiler.

- **Support Libraries**: A small number of libraries that must be linked with compiled models for aiding in Time Warp-based parallel simulation. Section 6 describes the interface provided by the libraries.

- **Support Tools**: A set of tools, such as the model-dependency generator and the resource database generator, as auxiliary support for model development and simulation. Section 7 describes these tools.

Section 8 lists some implementation constraints that result in a few deviations of the interface in this document from that described in the MetaTeD specification [1]. Section 9 lists the suggested sequence of steps for developing TeD models. Two detailed example models/applications are described in the appendices A and B.
2 Language Restrictions

The following restrictions apply to all C++ code physically included in TeD models as well as to any invoked code of libraries that are linked to the compiled models. In future versions of the interface, some of these restrictions may be either relaxed or supported in the form of an auxiliary library interface.

- All code must be reentrant. Static or global variables are not allowed anywhere.

- Input/Output operations, such as file operations, are not allowed — except for the purposes of printing to stdout/cout for debugging purposes or for printing results as part of result-wrapup.

- Dynamic memory allocation and deallocation are not permitted, with the following exceptions:
  - Deferred constants can be dynamically allocated in dconst-init (but never deallocated).
  - Large State items can be dynamically allocated as part of lstate-init (but never deallocated).
  - Result variables can be dynamically allocated in result-init.
  - At all other places, memory can be dynamically allocated and deallocated provided both the allocation and deallocation operations are performed during the same single sequential execution of the ext-statement.

- In the lstate construct, only the following data types are permitted. These types are designed to serve as replacements for the basic C++ data types. However, these types can be used in the normal fashion to form aggregates using any C++ compositional constructs such as arrays, classes and structures.

  - CHAR (designed to replace char)
  - INT (designed to replace int)
  - LONG (designed to replace long int)
  - DOUBLE (designed to replace double)

Note that the variables of lstate can be dynamically allocated as part of lstate-init.

- If the number and size of variables is large, then it is recommended that the variables be included in the lstate construct, instead of the state construct. Though using lstate entails changing the data types from int to INT, etc., it results in better performance when the sizes of the variables is “large” (more than 2Kbytes, for example).
3 Macro Interface

In order to make the argument type information clear, the argument syntax of each macro is written as though the macro were in fact a regular C++ function. A summary list of the macro signatures is given, followed by detailed description for each macro.

3.1 Macro Summary

The following “availability codes” are used in listing the different contexts of the external (C++) code in which each of the macros is available.

<table>
<thead>
<tr>
<th>Code</th>
<th>Available context</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE</td>
<td><code>ext-var-decls of event</code></td>
</tr>
<tr>
<td>VD</td>
<td><code>ext-var-decls of dconst</code></td>
</tr>
<tr>
<td>VS</td>
<td><code>ext-var-decls of state</code></td>
</tr>
<tr>
<td>VL</td>
<td><code>ext-var-decls of lstate</code></td>
</tr>
<tr>
<td>VR</td>
<td><code>ext-var-decls of result</code></td>
</tr>
<tr>
<td>XEC</td>
<td><code>ext-expr in entity channels declaration</code></td>
</tr>
<tr>
<td>XAC</td>
<td><code>ext-expr in architecture channels declaration</code></td>
</tr>
<tr>
<td>XPC</td>
<td><code>ext-expr in process array-channel variable specification</code></td>
</tr>
<tr>
<td>XPW</td>
<td><code>ext-expr in process wait statement</code></td>
</tr>
<tr>
<td>XCE</td>
<td><code>ext-expr in component entities definition</code></td>
</tr>
<tr>
<td>SDI</td>
<td><code>ext-statement of dconst init definition</code></td>
</tr>
<tr>
<td>SSI</td>
<td><code>ext-statement of state init definition</code></td>
</tr>
<tr>
<td>SLI</td>
<td><code>ext-statement of lstate init definition</code></td>
</tr>
<tr>
<td>SRI</td>
<td><code>ext-statement of result init definition</code></td>
</tr>
<tr>
<td>SRW</td>
<td><code>ext-statement of result wrapup definition</code></td>
</tr>
<tr>
<td>SFD</td>
<td><code>ext-statement of function definition</code></td>
</tr>
<tr>
<td>SPD</td>
<td><code>ext-statement of process definition</code></td>
</tr>
<tr>
<td>SCM</td>
<td><code>ext-statement of component map definition</code></td>
</tr>
<tr>
<td>SCW</td>
<td><code>ext-statement of component wrapup definition</code></td>
</tr>
<tr>
<td>Context</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>X^<em>, S^</em></td>
<td>int</td>
</tr>
<tr>
<td>X[<em>-EC], S^</em></td>
<td>T&amp;</td>
</tr>
<tr>
<td>SRW, SFD, SPD</td>
<td>T&amp;</td>
</tr>
<tr>
<td>SRW, SFD, SPD</td>
<td>T&amp;</td>
</tr>
<tr>
<td>SCW</td>
<td>T&amp;</td>
</tr>
<tr>
<td>SCW</td>
<td>T&amp;</td>
</tr>
<tr>
<td>SCW</td>
<td>T&amp;</td>
</tr>
<tr>
<td>SPD</td>
<td>T&amp;</td>
</tr>
<tr>
<td>SPD</td>
<td>bool</td>
</tr>
<tr>
<td>SPD</td>
<td>bool</td>
</tr>
<tr>
<td>SPD</td>
<td></td>
</tr>
<tr>
<td>SPD</td>
<td></td>
</tr>
<tr>
<td>SPD</td>
<td></td>
</tr>
<tr>
<td>SFD, SPD</td>
<td>double</td>
</tr>
<tr>
<td>SFD, SPD</td>
<td>void</td>
</tr>
<tr>
<td>SPD</td>
<td>bool</td>
</tr>
<tr>
<td>SPD</td>
<td>int</td>
</tr>
<tr>
<td>SPD</td>
<td>int</td>
</tr>
<tr>
<td>SPD</td>
<td>in-ch</td>
</tr>
<tr>
<td>VE</td>
<td>void</td>
</tr>
<tr>
<td>SCM</td>
<td>void</td>
</tr>
<tr>
<td>SCM</td>
<td>void</td>
</tr>
<tr>
<td>SDI</td>
<td>char*</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>bool</td>
</tr>
<tr>
<td>SDI</td>
<td>void</td>
</tr>
</tbody>
</table>
3.2 Macro Descriptions

The macros are listed here in alphabetical order, and their interfaces are described in detail.

<table>
<thead>
<tr>
<th>Macro</th>
<th>bool ACTIVE( in-channel ch, event-name evt )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To check if an event of type evt is active on an input channel variable, ch, of an entity (or any internal channel of an architecture).</td>
</tr>
<tr>
<td></td>
<td>event T1</td>
</tr>
<tr>
<td></td>
<td>{ {</td>
</tr>
<tr>
<td></td>
<td>int a;</td>
</tr>
<tr>
<td></td>
<td>CONSTRUCTOR( int b ) { a = b; }</td>
</tr>
<tr>
<td></td>
<td>} }</td>
</tr>
<tr>
<td></td>
<td>event T2</td>
</tr>
<tr>
<td></td>
<td>{ {</td>
</tr>
<tr>
<td></td>
<td>//...</td>
</tr>
<tr>
<td></td>
<td>} }</td>
</tr>
<tr>
<td></td>
<td>channel C { T1, T2 }</td>
</tr>
<tr>
<td></td>
<td>entity E</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>channels{ in C ch; }</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>arch A of E</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>behavior</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>process #1 p( ch );</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>process #1 E : A : p( ch )</td>
</tr>
<tr>
<td></td>
<td>{ {</td>
</tr>
<tr>
<td></td>
<td>if( ACTIVE( ch, T1 ) )</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>// Do something with event T1 on ch; say,</td>
</tr>
<tr>
<td></td>
<td>int x = DATA( ch, T1, a );</td>
</tr>
<tr>
<td></td>
<td>foo( x );</td>
</tr>
<tr>
<td></td>
<td>// ...</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Availability</td>
<td>In the <em>ext-statements</em> of process definition.</td>
</tr>
<tr>
<td>Macro</td>
<td><code>bool ACTIVE ANY( in-channel ch )</code></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Use</td>
<td>To check if an event of any type is active on an input channel variable, ch, of an entity (or any internal channel of an architecture).</td>
</tr>
</tbody>
</table>
| Example       | ```
entity E
{
    channels{ in C1 ch1; in C2 ch2; }
}
arch A of E
{
    behavior
    {
        process #1 p( ch1, ch2 );
    }
}
process #1 E : A : p( ch1, ch2 )
{
    if( ACTIVE_ANY( ch1 ) )
    {
        // There is some event on ch1
        // ...
    }

    if( ACTIVE_ANY( ch2 ) )
    {
        // There is some event on ch2
        // ...
    }
}
``` |
<p>| Availability   | In the <code>ext-statements</code> of <code>process</code> definition. |</p>
<table>
<thead>
<tr>
<th>Macros</th>
<th>int ASETSZ( in-channel-array ch )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int ASSETI( in-channel-array ch, int i )</td>
</tr>
<tr>
<td></td>
<td>in-channel ASETCH( in-channel-array ch, int i )</td>
</tr>
</tbody>
</table>

To find the **active set** (which is the set of all elements out of an array of input channels that have events active on them).

- ASETSZ() evaluates to the size of the active set (number of active channels in the array).
- ASSETI() evaluates to the index of the i'th element of the active set.
- ASETCH(ch,i) is an abbreviation for ch[ASSETI(ch,i)].

These macros are used for achieving better performance, compared to the alternative method of looping through all the elements of the channel array and using the ACTIVE_ANY macro to query their “active” status.

```c
entity E
{
    channels{ in C ch[$100$]; }
}
arch A of E
{
    behavior
    {
        process #1 p( ch[ $0$ to $99$ ] );
    }
}
process #1 E : A : p( ch[ $0$ to $99$ ] )
{
    for( int i = 0, n = ASETSZ(ch); i < n; i++ )
    {
        assert( ACTIVE_ANY( ASETCH(ch,i) ) );
        // or, equivalently,
        int j = ASSETI(ch,i);
        assert( ACTIVE_ANY( ch[j] ) );
        // ...
    }
}
```

**Availability**  In the *ext-statements* of *process* definition.
### Macro: CHANNEL

**Use**
To access any **output** channel variable of an entity (or any internal channel of an architecture) `ch`. 

**Example**
```c
event T
{
    int a;
    CONSTRUCTOR( int b ) { a = b; }
}
channel C { T }
entity E
{
    channels{ out C ch; }
}
arch A of E
{
    behavior
    {
        process #1 q;
    }
}
process #1 E : A : q
{
    // Send event T(10) on ch with delay 2.0
    CHANNEL(ch) << DELAY(2.0) << EVENT(T, (10));
    //...
}
```

**Availability**
In the `ext-statements` of `process` definition.

---

### Macro: void CONSTRUCTOR

**Use**
To define a constructor for an event.

**Example**
See the example for the `CHANNEL` macro.

**Availability**
In the `ext-var-decls` of `event`.

---

### Macro: T& DATA

**Use**
To access data of event named `evt` that is active on any **input** channel variable of an entity (or any internal channel of an architecture) `ch`.

**Example**
See the example for the `ACTIVE` macro.

**Availability**
In the `ext-statements` of `process` definition.
Macro $T$ & DCONST($T$ var )

**Use** To access a deferred constant item var.

```c
arch M of Mux
{
  dconst{ $double T; int B; }$
  state{ $int qlen;$ }
  behavior
  {
    process #i p;
  }
}

dconst Mux : M : init
{
  T = 1.0; // Emission time; default==1.0
  GET_RSRC_DOUBLE( "EmissionTime", &T );
  cout <<INAME() <<":EmissionTime" <<T <<endl;
  B = 200; // Buffer size; default==200
  GET_RSRC_INT( "BufferSize", &B );
  cout <<INAME() <<":BufferSize" <<B <<endl;
}

process #i Mux : M : p
{
  wait for $DCONST(T)$;
  { // And do something; say,
    if( STATE(qlen) == DCONST(B)-1 )
    {
      //...
    }
  }
}
```

**Availability** In all ext-exprs and ext-statements except ext-expr of entity channel declaration.

---

Macro DELAY( double period )

**Use** To add period as delay on an output channel.

**Example** See the example for the CHANNEL macro.

**Availability** In the ext-statements of process definition.
<table>
<thead>
<tr>
<th>Macro</th>
<th>EVENT( event-name evt, args-in-parens args )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To instantiate (in order to send) an event named evt which is constructed using args as arguments during construction. Note that parentheses are required around the second argument, which is the comma-separated argument list. Use the empty list, (), if the event has no constructor, or if its constructor takes no arguments.</td>
</tr>
<tr>
<td>Example</td>
<td>See the example for the CHANNEL macro.</td>
</tr>
<tr>
<td>Availability</td>
<td>In the ext-statements of process definition.</td>
</tr>
</tbody>
</table>

| Macros | bool GET_RSRC_STRING( char *rn, const char *ps [] )  
|        | bool GET_RSRC_INT( char *rn, int *pi )  
|        | bool GET_RSRC_DOUBLE( char *rn, double *pd )  
|        | bool GET_RSRC_INT_ARRAY( char *rn, const int *pa [], int *plen )  
|        | bool GET_RSRC_DOUBLE_ARRAY( char *rn, const double *pa [], int *plen ) |
| Use   | To get the value associated with the resource named rn for a particular entity instance. Each of the macros evaluates to TRUE if a resource of given name has been defined for the calling entity instance, FALSE otherwise. If no such resource is defined then the old value of the arguments are returned unmodified. For string and array valued resources, constant values are returned in the arguments, and hence the returned values cannot be modified by the caller. |
| Example | See the example for the DCONST macro. |
| Availability | In the ext-statement of init of dconst. |

<table>
<thead>
<tr>
<th>Macro</th>
<th>char * INAME( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>INAME() evaluates to the instance “name” of the entity, in dot-notation, starting with the root entity.</td>
</tr>
<tr>
<td>Example</td>
<td>See the example for the DCONST macro.</td>
</tr>
<tr>
<td>Availability</td>
<td>In the ext-statement of init of dconst.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro</th>
<th>T &amp; LSTATE( T var )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To access a “large state” variable var. Access method is similar to that of the DCONST macro.</td>
</tr>
<tr>
<td>Availability</td>
<td>In ext-exprs and ext-statements of process and function, and wrapup of result.</td>
</tr>
</tbody>
</table>
### Macro `void MAP( entity e1, channel ch1, entity e2, channel ch2 )`

**Use** To map the channel `ch1` of entity `e1` to the channel `ch2` of entity `e2`.

Note that `e1` and `e2` and/or `ch1` and `ch2` could be the same. Also, the mapping is symmetric (mapping `e1`'s `ch1` to `e2`'s `ch2` implies `e2`'s `ch2` is mapped to `e1`'s `ch1`), and hence the order of entities in the arguments is insignificant.

**Availability** In the `ext-statement` of `map` of `component`.

### Macro `void MAP_I( entity e1, int i1, entity e2, int i2 )`

**Use** To map the `i1`'th channel of entity `e1` to the `i2`'th channel of entity `e2`.

Note that `e1` and `e2` and/or `i1` and `i2` could be the same. This macro can be used to define entities that can contain an arbitrary collection of components whose channel interconnections can be deferred.

**Availability** In the `ext-statement` of `map` of `component`.

### Macro `double NOW( void )`

**Use** To find the value of current point on the timeline.

**Availability** In the `ext-statements` of `process` and `function`.

### Macro `int PARAM( int var )`

**Use** To access a parameter variable `var` of an entity or an architecture.

The implicit parameter, I (see [1]), is accessed like any other parameter — as `PARAM(I)`.

```c
entity E( int N )
{
    channels{ in C ch1[ $PARAM(N)$ ]; }
}
arch A( int M ) of E( int N )
{
    channels{ in C ch2[ $PARAM(M)$ ]; }
    behavior
    {
        process #1 p( ch1[0$ to $PARAM(N)-1$] );
        process #2 q( ch2[0$ to $PARAM(M)-1$] );
    }
}
process #1 E : A : p( ch[0$ to $PARAM(N)-1$] )
{
    // ...
}
```

**Availability** In all `ext-exprs` and `ext-statements`.
Macro

<table>
<thead>
<tr>
<th>Macro</th>
<th>void POT_ERR( const char *msg )</th>
</tr>
</thead>
</table>

Use

To register a potential error during the execution.

If an optimistic parallel simulator is used for simulating the models, it is often possible that an entity is pushed into a circumstance that is impossible to enter into if purely conservative/sequential simulation is performed. Any such circumstance is bound to be rolled back, unless instead it has been caused due to some programming error in the model itself. However, in general, it is difficult for the entity to distinguish a programming error from an error caused due to optimistic simulation. In such cases, this macro can be invoked at the location in the external code where the error is discovered. If the error is transient (caused by optimistic simulation), the error is automatically ignored; otherwise, the error persists (because it is associated with the model) and it is reported at the end of simulation.

Example

```c
process #1 E : A : p( ch[$0$ to $PARAM(N)$-1$] )
{
    // Suppose STATE(nactive) > 0 currently.
    for( int i = 0, n = ASETSZ(ch); i < n; i++ )
    {
        STATE(nactive)++;   // This situation could have happened
        if( STATE(nactive) > PARAM(N) )
        {
            // either due to optimistic simulation
            // or due to some programming error.
            POT_ERR( "#active can’t exceed N" );
        }
        else
        {
            // Do normal processing...
        }
    }
}
```

Availability

In the ext-statements of process and function.
Macros

bool PUT_RSRC_STRING( char comp[], char rn[], const char s[] )
bool PUT_RSRC_INT( char comp[], char rn[], int i )
bool PUT_RSRC_DOUBLE( char comp[], char rn[], double d )
bool PUT_RSRC_INT_ARRAY( char comp[], char rn[], const int a[], int len )
bool PUT_RSRC_DOUBLE_ARRAY( char comp[], char rn, const double a[], int len )

Use
To set the value associated with the resource named rn for particular component instance(s) of an architecture.
Note that resources can be specified only for immediate component instances (in other words, ‘.’ cannot appear in the component name parameter of the macros).
Each of the macros evaluates to TRUE if a resource of given name has been successfully set for the specified component(s), FALSE otherwise.

Example

class S of MuxSystem
{
    dconst( $int N;$ )
    behavior
    {
        components elems;
    }
}

components MuxSystem : S : elems
{
    entities
    {
        bsrc => Bursty : ? [$DCONST(N)$];
    }
    map
    {
        ...
    }
}

dconst Mux : M : init
{
    N = 10; // #elements in the system
    GET_RSRC_INT( "NumElements", &N );
    cout << INAME() << ":NumElements=" << N << endl;

    // Assume each source must be informed
    // about the total number of sources:
    PUT_RSRC_INT( "bsrc[.]", "N", N );
}

Availability
In the expr-statement of init of dconst.
<table>
<thead>
<tr>
<th>Macro</th>
<th>T &amp; RESULT( T var )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To access a result variable var of a component entity. Access method is similar to that of the DCONST macro.</td>
</tr>
<tr>
<td>Availability</td>
<td>In <em>ext-statements</em> of wrapup of component and wrapup of result.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro</th>
<th>void SET_PREFERRED_PE( int pe )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To register a preference of an entity/architecture instance for a given processor on a multiprocessor machine. This serves as a hint to the system to simulate the given entity on the processor with the given integer identifier.</td>
</tr>
<tr>
<td>Example</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>dconst E : A : Init</code></td>
</tr>
<tr>
<td></td>
<td><code>{</code></td>
</tr>
<tr>
<td></td>
<td><code>intppe = -1;</code></td>
</tr>
<tr>
<td></td>
<td><code>GET_RSRC_INT( &quot;PreferredPE&quot;, &amp;ppe );</code></td>
</tr>
<tr>
<td></td>
<td><code>SET_PREFERRED_PE( ppe );</code></td>
</tr>
<tr>
<td>Availability</td>
<td>In the <em>ext-statement</em> of init of dconst.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro</th>
<th>T &amp; STATE( T var )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To access a state variable var. Access method is similar to that of the DCONST macro.</td>
</tr>
<tr>
<td>Availability</td>
<td>In <em>ext-exprs</em> and <em>ext-statements</em> of process and function, and wrapup of result.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro</th>
<th>T &amp; SUBRESULT( entity comp, T var )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To access result variable var of a component entity.</td>
</tr>
<tr>
<td>Availability</td>
<td>In the <em>ext-statements</em> of wrapup of component.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro</th>
<th>T &amp; SUBRESULT_A( entity comp, int i, T var )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use</td>
<td>To access result variable var of a i'th element of a component in an array of component entities.</td>
</tr>
<tr>
<td>Availability</td>
<td>In the <em>ext-statements</em> of wrapup of component.</td>
</tr>
<tr>
<td>Macro</td>
<td>bool TIMEDOUT( void )</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Use</td>
<td>To check if a wait statement executed by a process has been timed-out. In such a wait statement, both an on clause and a for clause must be present. The check for timed-out condition must be made in the ext-statements immediately succeeding the wait statement and before any other succeeding wait statement.</td>
</tr>
<tr>
<td>Example</td>
<td>entity E</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>channels{ in C1 ch1; in C2 ch2; }</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>arch A of E</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>dconst{ $double T;$ }</td>
</tr>
<tr>
<td></td>
<td>behavior</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>process #1 p;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>process #1 E : A : p</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>wait on ch1, ch2 for $DCONST(T)$$;</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>if( !TIMEDOUT() )</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>assert(ACTIVE_ANY(ch1)</td>
</tr>
<tr>
<td></td>
<td>// ...</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Availability</td>
<td>In the ext-statements of process.</td>
</tr>
</tbody>
</table>
4 Configuration Language

The configuration language is used to specify the hierarchy of model instances, and customize them. The language allows the binding of a particular architecture to an entity type; the constructs of the language permit specification of different architecture bindings for different instances of the same entity type. In a similar vein, sufficient flexibility is provided for the customization of the resource values of architectures for different entity instances.

The configuration specification can be modified and customized independent of the model definitions. Thus, no recompilation of TeD models is necessary upon changing the configuration specification.

The configuration language consists of the following statements:

- Variable definition statement: Using this statement, variables and resource values can be defined using complex expressions.
- `for` statement: Using this statement, entity-architecture bindings and resource-value customization can be performed hierarchically.
- `require` statement: This statement facilitates the development of packages of predefined configurations of entity-architecture models.

The syntax and semantics associated with each of the statements are described next.

4.1 Variable Definition

```
tag type1 varname = value ;
```

- `type1` can be `int`, `double`, `string` or `name`. `type2` can be `int` or `double`.
- `value` can be an expression that evaluates to the type of the variable being defined.
- `int` expressions are formed using arithmetic operators `+`, `-`, `*`, `/` and `%`. `double` valued expressions can be converted to `int` using the `int()` cast function.
- `double` expressions are formed using the same operators as for `int`, except for the `%` operator. `int` values can be converted to `double` using the `double()` cast function.
- A `double` constant must contain enough information (a period or the letters `e` or `E`) to distinguish it from a pure `int` constant.
- `string` constants are formed by enclosing any text between a pair of "s.
- `name` expressions are formed according to the following grammar:
4 CONFIGURATION LANGUAGE

| name_expr: name_term | name_expr ',' name_term
| name_term: name_item arr_spec
| name_item: word | '0' name_var
| arr_spec: empty | '[' low_spec ',' high_spec ']'
| low_spec: empty | int_expr
| high_spec: empty | int_expr

An empty low_spec implies the index 0. An empty high_spec implies the index of the last element of the array. When a name variable is used as @name_var in a name expression, the value of the name_var is substituted in the place of @name_var.

- **arrays**: is optional. If unspecified, the size of the array is the number of items in the comma-separated value list. If specified, the number of items in the value list must match the given array size.

- **tags** can be combinations of meta, resource and override. The meta tag can be used only if the resource tag is also used, and only if the varname is an array variable. name variables cannot be tagged as a resource. override tag is used to assign a different value to an already defined variable. It is an error to redefine a variable without using the override tag.

- If the resource tag is specified for a variable, then a resource with the given type, name and value is “generated” for the entity instance(s) that is(are) specified by the immediately enclosing for statement.

- If the meta tag is used for an array resource, and the enclosing for statement specifies an array of entities, then the following effect is achieved: the i'th value of the array resource will map to the resource of the i'th entity; each of those values will be assigned to a resource whose name is the same as the name of the array variable. This is useful in easily defining different individual values for the same resource name for an array of entities. Achieving the same effect, without the meta tag concept, is possible but tedious (using separate for statements for each individual element of the array of entities).

4.2 For Statement

```plaintext
for name-expression [use entity-type( parameters ) : arch-type( parameters )]
{
   for, require or variable-definition statements
}
where, parameters is a comma-separated list of integer expressions.
```

The for statement is used to specify or customize the resource values for one entity or an array of entities. The set of entities are specified using the name-expression. The name-specification of the entities is relative to the immediately enclosing entity or entities as specified by the immediately enclosing for statement, if any. For example, consider the following TeD model.
Further, consider the following configuration specification that uses an instance of $RE:RA$ as the root entity.

```c
text RE
{
  channels{ ... }
}
arch RA of RE
{
  dconst{ $int N;$ }
  behavior
  {
    components Z;
  }
}
components RE : RA : Z
{
  entities
  {
    comp => ? [ $DCONST(N)$ ];
  }
  //...
}
```

The preceding configuration specifies that $RE:RA$ be used as the root entity, and $SE:SA$ be used for its array of subentities named `comp`. Thus, the resource $int dummy$ applies to the array of entity instances implied by the name `root.comp[0,N-1]`.

The `use` clause may be omitted if a default entity-architecture pair has been specified for the corresponding component in the TeD model. For example, the component definition and configuration specifications could be modified as follows:

```c
text RE
{
  entities
  {
    comp => SE : SA [ $DCONST(N)$ ];
  }
}
```
for root use RE() : RA()
{
    resource int N = 100;
    for comp[0,N-1]
    {
        resource int dummy = 10;
    }
}

Variable definitions are scoped as follows: Variables defined inside a for statement are visible in expressions from the point of definition to the end of the enclosing for statement. Variables that are defined outside the top-most for statement(s) are visible as though there was an imaginary enclosing for statement containing all the variable definitions and top-most for statement(s).1

4.3 Require Statement

require type1 var-name;
or
require type2 var-name [];

Type1 can be int, double, string or name. type2 can be int or double.

The require statement is used to assert that a variable of the given name and type has been defined and is visible in the same scope as that of the require statement. This statement is useful in developing packaged configurations of models. This is done by specifying assumptions of certain values that are required in the configuration. For example, a require int N; statement may be placed at the top of a configuration specification for a multiplexer.

4.4 Preprocessing

The contents of configuration files are passed through a C++ preprocessor, and the resulting output is presented to the resource database generator program. This feature can be effectively exploited to achieve packaged models as follows: For modeling each entity type E, three files — E.the, E.ted and E.cfg — are used. All the entity and architecture declarations are specified in the file E.the. Its associated process and component definitions, dconst-init, etc., are included in E.ted. The various resource definitions with default constant values are specified in E.cfg. Given such an organization, suppose another user needs to use E as a component, comp, of another entity F’s architecture. For this, the statement use "E" is placed in F.the. In the configuration file of F, F.cfg, the following statement is included:

```cpp
// Assume EA is an arch of E
for comp use E() : EA()
{
    #include "E.cfg"
}
```

1Such variables must not be tagged as resource items.
If the user wishes to customize the configuration of $E$ by overriding a resource definition of $E$, the `override` tag can be used on the redefinition of that resource and placed after the `#include "E.cfg"` statement.

5 Compiler

The compiler takes TeD models as input and generates C++ code that can be compiled using a regular C++ compiler to produce an executable. Using the generated executable along with the resource database generator (see Section 7), the TeD models can be assembled, customized, and simulated.

The class names and associated code in the generated C++ target are designed for easy readability so that the naming is useful during debugging sessions using conventional debuggers.

The compiler is available for the Solaris operating system environment on shared-memory Sun Sparc systems, and for the Irix 6.x operating system environment on the SGI PowerChallenge and SGI Onyx shared-memory machines. It has been tested with the CC and g++ compilers.

Usage

If a filename is specified on the command line, the name must have a suffix of .the or .ted. If no filename is specified, then stdin is parsed, and output is generated to stdout. When stdin is parsed, the compiler behaves as though it is parsing a .ted file.

In addition, the following options are accepted by the ted compiler.

- `-l` : By default, the compiler generates `#line` directives in the output C++ code for easy mapping of compiler reports to the corresponding line numbers in the TeD model files. Using the `-l` option, this feature can be turned off.

- `-o filename` : This places the generated output code in a file of given name.

Implementation

Figure 1 shows the layered approach to the design and implementation of the compiler for TeD.

6 Support Libraries

- **GTW**: GTW is the C-language interface to the Georgia Tech Time Warp parallel discrete event simulation system, as described in [3].

- **GTW++**: GTW++ is a C++-language interface to GTW. In addition to the parallel simulation service, this interface provides several useful random number generator
classes that can be used in TeD models. The C++ header file for accessing the GTW++ library is automatically included in the compiled target code of the models, and hence the classes are directly visible in all the places where C++ code can appear in the TeD models. The class interfaces for the random number generators are described in [2].

- **GTWTED**: GTWTED is a class library that is built on top of GTW++. It defines classes that support the concepts of entities, architectures, processes, components and other TeD concepts. The classes make use of the GTW++ class interface for simulating the behavior of the entities. The transparent incremental state saving interface described in [4] is used in GTWTED for supporting the large-state concept in TeD.

7 Support Tools

The tools provided along with the software release include the following:

- Model Dependency Generator (**tedmake**)
- Resource Database Generator (**tedrdbgen**)
- Model Information Collector (**tedcollect**)
- Platform-independent System Builder (**tedaimk**)

Sample **Makefile** and **makefile.aimk** are provided along with the example applications in the TeD software release. They are designed so that the compiler and the tools get invoked automatically.
7.1 Model Dependency Generator

The Model Dependency Generator (tedmake) takes several TeD model and configuration files as input. It detects interdependencies among them and generates a file in which the dependencies are specified in a format suitable for feeding to the make program. By default, tedmake generates the dependencies into a file named ted.depend.

tedmake detects dependencies of .ted and .the files based on the use and import statements appearing in those files. In .h, .c, .cfg and any other files, it detects dependencies based on directives of the form #include "filename". It ignores directives of the form #include <filename>. It only looks in the current working directory when trying to find any file.

7.2 Resource Database Generator

The Resource Database Generator (tedrdbgen) takes configuration specifications expressed in the Configuration Language (see Section 4) and generates a “flattened” output, called the resource database, that is in a format suitable for feeding to the model simulation executable. The file model.cfg is the default input file, and model.rdb is the default target filename. The resulting resource database must be passed to the model executable using the command line arguments of the executable.

7.3 Model Information Collector

The Model Information Collector (tedcollect) is used to gather entity instantiation information from the C++ code of the compiled models. The Compiler places such information in the generated code in a form that is easily recognizable by tedcollect. The gathered information is assembled by tedcollect into a file named .collect.C and the file is automatically compiled along with the C++ code of the models.

The collected information is used during the instantiation of the entities and in binding architectures in a later-than-compile-time fashion. The instantiations and bindings are performed according to the configuration specification.

7.4 Platform-independent System Builder

The Platform-independent System Builder (tedaimk) is useful in the case of a shared file system in which the same set of files are shared by several computers of different architectures. This tool is used to automatically select the right set of executables and options throughout the model development process.

The underlying mechanism of architecture-independent build process is based on the automatic assignment of a distinct code value that depends on the architecture on which tedaimk is executed. A fixed architecture code is assigned to each distinct type of machine architecture. This code is used in selecting the correct set of files in the various phases of the build process.
8 Deviations from MetaTeD

- Simultaneous events are not presented simultaneously to processes. Instead, they are presented one after the other sequentially.

- Both entity and architecture types cannot be inherited simultaneously. For example, the following is not possible in the current implementation: If $E_1$ is an entity type, and $A_1$ is an architecture of $E_1$, then, another entity $E_2$ inherits entity type $E_1$, and an architecture $A_2$ of $E_2$ inherits from $A_1$.
  However, the following are possible: $E_2$ inherits $E_1$ but no architecture $A_2$ of $E_2$ inherits from any architecture $A_1$ of $E_1$. Or, an architecture $A_2$ of $E_2$ inherits from any other architecture $A$ of $E_2$.

- In a wait statement, the until clause can appear only with the on clause — it cannot appear all by itself, and it cannot be combined with the for clause.

- A limit of 20 processes exists per entity.

- A limit of 20 wait statements exists per process.

- wait statements are not allowed inside ext-statements of a process.

- wait statements are not allowed inside functions.

- No provision exists to “redirect” events on an interface channel of an entity to the interface channel of a component entity\(^2\). In other words, it is not possible (in a straight-forward way\(^3\)) to map an interface channel simultaneously to another entity’s interface channel as well as a component’s interface channel.

---

\(^2\) This is due mainly to the limitation of at most one channel that can be mapped to any given channel. Mapping a pair of channels to each other results in a closed ring, rather than a chain, of channels.

\(^3\) An alternative, albeit tedious, way is to use a separate process to forward events arriving on the interface channel to the component’s channel.
9 Model Development Guidelines

The following is the suggested sequence of steps for developing a set of models for any application. Suppose the application is called MYAPP.

1. Create a directory, say MYAPP. It is not necessary that the directory be created under the TeD installation directories. All the application's files, as specified in the following steps, must be created in this new directory MYAPP.

2. Copy into the MYAPP directory the Makefile and makefile.aimk of any example TeD application that is provided along with the TeD software release. Edit the Makefile, if necessary, to point the variable TEDITINSTALLDIR to the correct location of the TeD installation directory according to your local system setup.

3. For every set of logically related event types and channel types, create a file with .the extension, and declare all those types in that file.

4. For each entity type, E, perform the following:
   
   (a) Create a corresponding file, say E.the, and declare the entity and architecture types in that file. Use the use statement to include the .the files that contain the event and channel types.
   
   (b) Define the dconst-init, state-init, istate-init, processes, components, etc. in a file named E.ted. Again, use the use statement to include the .the files that contain the entity and architecture declarations.
   
   (c) Place all model-dependent C++ class declarations in .h files and use the import interface statement to use them in the TeD models. Place all model-dependent C++ method/function definitions in .c files, and use the import implementation statement to include the definitions in a .ted file.
   
   (d) (Optional step) Create a file, E.cfg, in which the default resource values for the architectures of the entity are specified using the Configuration Language.

5. Create a file, model.cfg, that contains the model configuration specification starting with the root entity. The default configurations of the various models, as given in their .cfg files, can be utilized in model.cfg using the #include C++ preprocessor directive.

6. Type make to initiate a complete build of the application. (Type make nolines whenever it is desired to turn off the generation of #line directives in the generated C++ output).

   The Makefile and makefile.aimk together automatically invoke the Model Dependency Generator (tedmake), the TeD compiler (ted), the Model Information Collector (tedcollect) and the Resource Database Generator (teddbggen). The build process itself initially starts by the invocation of the Platform Independent System Builder
(tedaimk) which in turn invokes the executables corresponding to the architecture of
the computer on which the system is being built.

7. Upon successful completion of the build process, the model configuration is executed
by typing the following in the MYAPP directory:

    OBJ/ARCH/runmodel p T -A model.rdb

where, ARCH is the architecture code of the computer, p is the number of processors
to use for simulation, and T is the total amount of simulated time to simulate.

References

[1] “MetaTeD — A Meta Language for Modeling Telecommunication Networks,” K. S. Pe-
rumalla, A. T. Ogielski, R. M. Fujimoto, Technical Report GIT-CC-96-32, College of
Computing, Georgia Institute of Technology, 1996.

System,” K. S. Perumalla, R. M. Fujimoto, Technical Report GIT-CC-96-09, College of
Computing, Georgia Institute of Technology, 1996.


College of Computing, Georgia Institute of Technology, 1996.
A  Example 1: Pingpong

Figure 2: Pingpong example

Figure 2 depicts the Pingpong model that will be used as a simple example to illustrate and put together the various concepts (including inheritance and process-overloading) in TeD\footnote{This is a contrived example, and does not correspond to any real life application; it is intended to serve as a very simple application that does not require the reader to have a lot of background information, and yet covers most of the concepts.}.

In this example, two players — Ping and Pong — play against each other. “Playing” implies that whenever a player receives a ball, it simply responds by sending another ball to the other player. The two players behave exactly the same, except that Ping serves the very first ball. Each player is interested in collecting statistics on the number of balls received and the number of balls sent by itself.

The application is modeled in TeD as follows. A Player is defined as an entity with one input BallChannel and one output BallChannel. Ball events can be sent or received on the channels. Two different architectures — one each to model Ping and Pong — are defined for the Player entity. Since Ping and Pong are almost exactly similar, Pong is defined to inherit all of Ping’s behavior, but it overrides that part of the behavior in which it differs (the process definition).

First, the Ball event is declared, and the BallChannel channel type is declared in a file named ball.the as follows.

```latex
event Ball { $\text{No data} \}$
channel BallChannel { Ball }
```

Then, the Player entity, which is the external view of each of the players, is defined in
a file named `player.the` as follows:

```plaintext
entity player
{
    channels
    {
        in BallChannel A;
        out BallChannel B;
    }
}
```

The behavior of `Ping` is defined, in `player.the`, as an architecture of the `Player` entity as follows:

```plaintext
arch ping of player
{
    dconst{ $ double travel_time; }$ 
    state { $ int nrecd; int nsent; }$ 
    behavior
    {
        process #1 hit;
    }
    result { $ int nrecd; int nsent; }$ 
}
```

In the above, the time that a ball takes to travel from `Ping` to `Pong` is defined as a deferred constant in the definition of `Ping`. A process, `hit`, is declared, and it will be defined later in the section.

The behavior of `Pong` is defined in terms of that of `Ping`, since they are exactly similar except for the difference of who serves first. This is achieved by declaring, in `player.the`, the architecture `Pong` to `inherit` from the architecture `Ping`, but overriding the `hit` process, which will be defined differently.

```plaintext
arch pong of player inherit ping
{
    behavior
    {
        process #1 hit( A ) overload;
    }
}
```

The process `hit` for the two players is defined in a file named `player.ted` as follows:
example 1: pingpong

```
process #1 player : ping : hit
{
  $ CHANNEL( B ) << DELAY( travel_time ) << EVENT( Ball, () ); $
  $ STATE(nsent)++; $
  wait on A;
  $ STATE(nrecd)++; $
}

process #1 player : pong : hit( A )
{
  $ STATE(nrecd)++; $
  $ CHANNEL( B ) << DELAY( travel_time ) << EVENT( Ball, () ); $
  $ STATE(nsent)++; $
}
```

The hit process of **Ping** does not have any channels on its sensitivity list, and so, its statements are executed straight away. In its first statement, it sends a **Ball** event on its output channel **B**, and increments its **nsent** counter in the second statement. It then waits for a response **Ball** to arrive on its input channel **A**. The statement after the wait statement is executed as soon as an event arrives on the channel **A**. At the end of execution of the last statement, the process loops around to the first statement, and hence, the above process is repeated.

In contrast, the hit process of **Pong** is defined to be sensitive to the input channel **A**. This implies that the set of statements of this process are executed whenever there is an arrival on the channel **A**. Thus, this process remains dormant until a **Ball** arrives on **A**. When the **Ball** does arrive, then all the three statements of the process are executed one after the other, and then the process loops back to dormant state again, waiting for another arrival on **A**.

The initialization of deferred constants is defined in **player.ted** as follows:

```
dconst player : ping : init
{
  
  travel_time = 1.0; // default value
  GET_RSRC_DOUBLE( "BallTravelTime", &travel_time );
  cout << INAME() << ":BallTravelTime=\" << travel_time << endl;
}
```

The initialization of state is defined in **player.ted** as follows:

```
state player : ping : init
{

  nsent = nrecd = 0;

}
```

The initialization and wrapup of result is defined in **player.ted** as follows:
A  EXAMPLE 1: PINGPONG

```c
result player : ping : init
{\{
    nsent = nrecd = 0;
\}}

result player : ping : wrapup
{\{
    nrecd = STATE(nrecd); nsent = STATE(nsent);
    cout << "Player: nrecd = " << nrecd
    << ", nsent = " << nsent << endl;
\}}
```

Now, a new entity type Game is declared, in order to model the game. The “behavior” of this game entity is described as a composition of the two players. The two component players will be interconnected appropriately so that they “play against each other.” The Game entity and its PingPong architecture are defined in a file named game.thf as follows:

```c
entity game
{
    channels { }
}

arch pingpong of game
{
    behavior
    {
        component players;
    }
}
```

The behavior of the Game is defined in terms of its component players, which is defined in a file named game.ted as follows:

```c
components game : pingpong : players
{
    entities
    {
        ping => player : ping;
        pong => player : pong;
    }
    map
    {\{
        MAP( ping, A, pong, B );
        MAP( ping, B, pong, A );
    \}}
    wrapup
    {\{
        cout << "SUBRESULT(ping,nsent) == "
        << SUBRESULT(ping,nsent) << endl;
    \}}
```

The preceding component description instantiates Ping and Pong as part of the entities
A  EXAMPLE 1: PINGPONG

part of the description. The map section of the component maps the input channel of A to
the output channel of B, and vice versa. The wrapup section of the description is executed
after the component entities have completed their simulation. Thus, the number of balls
sent out by the Ping entity is printed out as the result.

Finally, the whole game is instantiated using the following configuration specification in
a file named model.cfg (with some arbitrary set of resource values):

```plaintext
for root use game() : pingpong()
{
    for ping
    {
        resource double BallTravelTime = 3.0;
    }
    for pong
    {
        resource double BallTravelTime = 7.0;
    }
}
```
B Example 2: ATM Multiplexer

Figure 3: Multiplexing system example

Consider the description of a simple ATM multiplexer operating on a set of input sources. The sources emit ATM cells, and the multiplexer multiplexes the cells coming from the sources onto a single output link. The multiplexer has an internal buffer in order to hold cells under heavy input traffic conditions. The measures of interest are the number of cells transferred to the output link, and the number of cells lost (dropped) in the multiplexer due to buffer overflows. The use of `lstate` construct is illustrated by using it to measure cell delay distribution.

Cell Event and Cell Channel

First, a channel type is defined that specifies all the types of data that can flow across any channel of that type. One type of data unit – `ATMCell` – is declared, and a channel of type `ATMChannel` is declared that allows `ATMCells` to flow through it. These declarations are placed in a file named `call.the`.

```plaintext
event ATMCell
{
    // No data variables used.
    CONSTRUCTOR( void ) {}
}

channel ATMChannel { ATMCell }
```
Multiplexer

The multiplexer entity is defined, in a file named \texttt{mux.the}, as one with \texttt{N} input channels, each of type \texttt{ATMChannel} (in this example, we will not model the generation of output cell stream from the multiplexer; however, it can be added in a straightforward way to the code in this example):

```plaintext
entity Mux( int N )
{
    channels
    {
        in ATMCellChannel A[ $PARAM(N)$ ];
    }
}
```

Note that \texttt{N} is a parameter to the entity, implying that the value of \texttt{N} has to be specified at the time of instantiating the entity.

One (of several possible) descriptions of the behavior for the multiplexer is declared in \texttt{mux.the}. This architecture keeps track of the number of transferred cells and the number of lost cells. It uses two processes to model the multiplexer's behavior.

```plaintext
architecture Mux of Mux( int N )
{
    dconst { $int B, C; double T;$ }
    state { $int qlen; int nsent; int nlost;$ }
    lstate { $INT *qfreq;$ }
    behavior
    {
        process #1 scan( A );
        process #2 update;
    }
    result { $int nsent; int nlost;$ }
}
```

Note that \texttt{B}, denoting the buffer size, and \texttt{C} denoting the factor by which the output link of the multiplexer is faster as compared to each of the input links, and \texttt{T} denoting the time for one cell emission on any input link, are declared as deferred constants of this architecture. Also, note the use of \texttt{lstate} containing a variable \texttt{qfreq} that will be used to keep track of cell-delay distributions.

The deferred constants are initialized in \texttt{mux.ted} as follows:
dconst Mux : Mux : init
{\}
  B = 5; // default value
  GET_RSRC_INT("BufferSize", &B);
  cout << INAME() << ":BufferSize=" << B << endl;

  C = 5; // default value
  GET_RSRC_INT("LinkCapacity", &C);
  cout << INAME() << ":LinkCapacity=" << C << endl;

  T = 1.0; // default value
  GET_RSRC_DOUBLE("EmissionTime", &T);
  cout << INAME() << ":EmissionTime=" << T << endl;
}\}
The state variables are initialized in mux.ted as follows:
state Mux : Mux : init
{\}
  qlen = 0; nsent = 0; nlost = 0;
}\}
The lstate items are allocated and initialized in mux.ted as follows:
lstate Mux : Mux : init
{\}
  qfreq = new INT[ DCONST(B)+1 ];
  for( int i = 0; i < DCONST(B)+1; i++ ) qfreq[i] = 0;
}\}
Note the use of dynamic memory allocation to set the lstate variables.
The two processes of the multiplexer are defined in a file named mux.ted as follows.

process #1 Mux : Mux : scan( AI $0$ to $PARAM(N)-1$ )
{\}
  for( int i = 0, n = ASETSZ(A); i < n; i++ )
  {
    assert( ACTIVE( ASETCH(A,i), ATMCell ) );

    int curr_len = STATE(qlen);

    if( curr_len >= DCONST(B) )
    {
      STATE(nlost)++;
      LSTATE(qfreq[DCONST(B)])++;
    }
    else
    {
      LSTATE(qfreq[curr_len])++;
      STATE(qlen)++;
    }
  }
}\}
The first process scan is executed whenever at least one of the input channels has a data arrival on it.
process #2 Mux : Mux : update
{
  \{ if( STATE(qlen) > 0 )
  \{
    STATE(qlen)--;
    STATE(nsent)++;
  \}
  \} wait for $DCONST(T)/DCONST(C)$;
}  

The second process update is scheduled to be executed every one cell emission time period. This process calculates and updates the state information defining the queue length and the number of cells lost so far\(^5\).

The result variables are initialized and used in mux.ted as follows:

```c
result Mux : Mux : init
{\{
  nsent = 0; nlost = 0;
}\}}

result Mux : Mux : wrapup
{\{
  nsent = STATE(nsent); nlost = STATE(nlost);
  cout << "MUX: nsent=" << nsent << ",nlost=" << nlost << endl;
  cout << "MUX: Delay distribution = {" << endl;
  for( int i = 0; i < DCONST(B)+1; i++ )
  {   
      double prob = LSTATE(qfreq[i]) / double( nsent+nlost );
      cout << " " << prob << endl;
  }
  cout << " }" << endl;
\}}
```

Cell Source

A data source entity is declared that emits cells as its output stream. Instances of this type of entity will be used as inputs to the multiplexer. The BurstySource entity is defined, in a file named source.thc, to have zero input channels, and exactly one output channel of type ATMChannel:

```c
entity BurstySource
{
  channels { out ATMChannel A; }
}
```

One interesting type of data source is an ON/OFF data source, which continuously emits cells if it is in an ON state, and does not emit any cells if it is in an OFF state. The source

\(^5\)Since the semantics of concurrent processes state that two processes, if they are active simultaneously, are executed as though they were executed sequentially one after the other, no conflict conditions arise due the read/write accesses to the shared state variable qlen.
switches between the ON and OFF states with some given probabilities, which are specified as parameters to this type of behavior.

```vhdl
architecture UniffSource of BurstySource
{
  dconst( $double T; long seed; double p_on, p_off;$ }
  state{
    CBool am_active;
    CUniformRNG active;
    CUniformRNG silent; 
  }
  behavior
  {
    process #i emit;
  }
}
```

The preceding behavior declaration, in `mix.the`, uses one process named `emit` that periodically updates the source’s state every one cell emission time. For every update, the source calculates its next state, using the ON/OFF probabilities. If it turns out to be in an ON state, then it emits one cell on its output channel.

The `emit` process of the `OnOffSource` is defined in a file named `mix.ted` as follows:

```vhdl
process #i BurstySource : UniffSource : emit
{
  \{ 

    if( STATE(am_active) )
      STATE(am_active) = STATE(active).next() > DCONST(p_on);
    else
      STATE(am_active) = STATE(silent).next() > DCONST(p_off);

    if( STATE(am_active) )
      CHANNEL(A) << DELAY(DCONST(T)) << EVENT(ATMCell, ());
  \}
  wait for $DCONST(T$;
}
```

The deferred constants are initialized in `mix.ted` as follows:
dconst BurstySource : UnifSource : init
{
    // Default values
    T = 1.0;
    seed = PARAM(I)*10 + 10;
    p_on = 0.5;
    p_off = 0.9;

    GET_RSRC_DOUBLE( "EmissionTime", &T );
    cout << INAME() << ":EmissionTime=" << T << endl;

    int int_seed;
    if( GET_RSRC_INT( "Seed", &int_seed ) ) seed = int_seed;
    cout << INAME() << ":Seed=" << int_seed << endl;

    GET_RSRC_DOUBLE( "P_on", &p_on );
    cout << INAME() << ":P_on=" << p_on << endl;

    GET_RSRC_DOUBLE( "P_off", &p_off );
    cout << INAME() << ":P_off=" << p_off << endl;
}

// state is initialized in mux.ted as follows:
state BurstySource : UnifSource : init
{
    am_active = FALSE;
    CRNGSeed s( DCONST(seed), 2*DCONST(seed) );
    active.init( s ); silent.init( s );
}

Multiplexing System

We proceed to using the preceding two types of entities, and create one aggregate entity
that consists of the sources as well as the multiplexer:

entity MuxSystem
{
    channels {}
}

architecture MuxSystem of MuxSystem
{
    dconst { $int N;$ } behavior
    {
        component elems;
    }
}

The preceding multiplexing system entity and its architecture declarations are placed in
a file named system.ted. The architecture of the multiplexing system is defined in terms
of its components. The components of the multiplexing system are defined in a file named
B  EXAMPLE 2: ATM MULTIPLEXER  38

system.ted as follows.

```c
component MuxSystem : MuxSystem : elem
{
    entities
    {
        mux => Mux %i( $DCONST(N)$ ) : Mux;
        src => BurstySource : OnOffSource [ $DCONST(N)$ ];
    }

    map
    {
        for( int i = 0; i < DCONST(N); i++ )
            MAP( mux, A[i], src[i], A );
    }

    wrapup
    {
        cout << "MuxSystem: nsent = " << SUBRESULT(mux,nsent);
        cout << ", nlost = " << SUBRESULT(mux,nlost) << endl;
    }
}
```

Note that, the behavior of the multiplexing system is described in terms of its constituent parts – N sources, and one multiplexer.

Finally, the startup configuration is defined in a file named model.cfg as follows.
```c
int MAX_N = 18;
int N = 10;
double gain = 3.0;
double T = 1.0;

for root use MuxSystem() : MuxSystem()
{
    resource int NumInputs = N;
    
    for mux
    {
        int K = int( double( N ) / gain + 0.5 );
        
        resource int BufferSize = 2*K;
        resource int LinkCapacity = K;
        resource double EmissionTime = T;
    }

    name allsources = src[0,N-1]; // For readability
    for allsources
    {
        resource double EmissionTime = T;
        resource double P_on = 0.6;
        resource double P_off = 0.8;
        meta resource int Seed[MAX_N] =
            { 12, 29, 340, 978, 7634, 13, 09, 872,
              82, 987, 23, 897, 623, 734, 8973 };
    }
}
```

This specifies the root entity of the application is the multiplexing system, with some specified values as parameters for the system. Note the use of the `meta` tag to specify different seed values for each of the sources.